

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ПЛАТФОРМА MICROSOFT.NET И ЯЗЫК ПРОГРАММИРОВАНИЯ

C#

Урок №3

Структуры.
Классы.
Свойства

Содержание

1. Структуры	5
Понятие структуры.....	5
Синтаксис объявления структуры	5
Необходимость и особенности применения структур.....	7
Конструктор без параметров и структуры	7
2. Синтаксис объявления класса	10
3. Модификаторы доступа языка программирования C#.....	13
4. Поля класса	15
5. Конструкторы.....	19
Понятие конструктора.....	19
Параметризованный конструктор	22

Перегруженные конструкторы.	23
Статический конструктор.	25
6. Ключевое слово <code>this</code>.	28
7. Методы класса.	32
Передача параметров.	34
Ключевое слово <code>return</code>	36
Перегрузка методов.	38
8. Использование <code>ref</code> и <code>out</code> параметров.	40
Использование модификатора <code>ref</code>	42
Использование модификатора <code>out</code>	44
9. Создание методов с переменным количеством аргументов.	46
10. Частичные типы (<code>partial types</code>).	48
11. Свойства.	51
Что такое свойства?.	51
Синтаксис объявления свойств.	53
Примеры использования свойств.	60
Автоматические свойства.	64
Что такое автоматические свойства?.	64
Инициализация автоматических свойств.	66
Примеры использования автоматических свойств.	69
Null-conditional оператор.	71
12. Пространство имён.	75
Что такое пространство имён?.	75
Цели и задачи пространства имён.	77

Ключевое слово <code>using</code>	85
Объявление пространства имён	90
Вложенные пространства имен	90
Разбиение пространства имён на части.....	94
Пространство имён по умолчанию.....	96
Директива псевдонима <code>using</code>	97
Использование <code>using</code> для подключения статических членов	101
13. Домашнее задание	103

1. Структуры

Понятие структуры

Структура — это составной тип данных, который является последовательностью переменных различных типов. В C#.NET структуры предназначены для группирования и хранения небольших порций данных.

Структуры — это типы значений и наследуются от `System.ValueType`. Это значит, что они либо сохраняются в стеке, либо являются встроенными (последнее — если они являются частью другого объекта, хранимого в куче), и имеют те же ограничения времени жизни, что и простые типы данных.

Синтаксис объявления структуры

Для объявления структуры используют ключевое слово `struct`:

```
struct Имя : Интерфейсы
{
    //объявления членов
}
```

Приведем пример работы со структурой. Напишем программу, которая будет, работать со структурой, описывающей какие-то габаритные размеры.

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

В структурах возможно определение полей, свойств, методов, конструкторов, итераторов и событий (некоторые из них Вы будете изучать позже). Например:

```
struct Dimensions
{
    private double Length;
    public double Width;

    public Dimensions(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public void Print()
    {
        Console.WriteLine($"Длина {Length}, ширина {Width}.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        double length = 7.342, width = 23.49;
        Dimensions dimensions =
            new Dimensions(length, width);
        dimensions.Print();
    }
}
```

Результат работы программы (Рисунок 1.1):

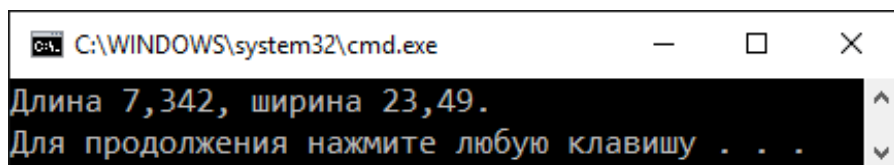


Рисунок 1.1. Пример создания структуры

Необходимость и особенности применения структур

Структуры используют для оптимизации производительности, когда из функциональности класса нужно только хранение некоторых небольших порций данных. Структуры можно трактовать как упрощенные классы.

Тип `struct` не требует отдельной ссылочной переменной. Это означает, что при использовании структур расходуется меньший объем памяти, и благодаря прямому доступу к структурам, при работе с ними не снижается производительность, что имеет место при доступе к объектам классов. За счет хранения в стеке выделение и удаление памяти под структуры происходит очень быстро.

При присваивании одной структуры другой или при передаче методу структуры, как параметра, происходит полное копирование содержимого структуры, что намного медленнее передачи ссылки. (Скорость копирования зависит от размера структуры: чем больше полей — тем дольше копирование).

Структуры имеют доступ к методам класса `System.Object`, которые могут переопределять. Структуры не поддерживают наследование. Структуры могут наследовать интерфейсы.

Конструктор без параметров и структуры

Компилятор всегда генерирует конструктор по умолчанию без параметров, который переопределить невозможно. Выделение памяти под всю структуру происходит при объявлении переменной. Соответственно, операция `new` для структур действует иначе, чем для ссылочных

ТИПОВ: она вызовет инициализацию полей значениями по умолчанию.

```
public void DoSomething()
{
    Dimensions d;
    //память выделена, но поля значениями
    //не инициализированы;
    d = new Dimensions();
    d.Print(); //Length равно 0, Width равно 0;
    d.Length += 2;
    d.Width += 4;
    d.Print(); //Length равно 2, а Width равно 4;
}
```

Результат выполнения кода представлен на рисунке 1.2:

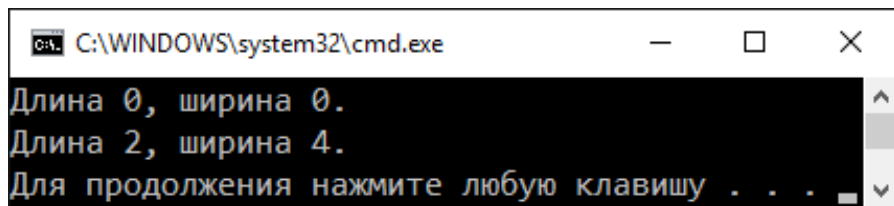


Рисунок 1.2. Пример работы со структурой

Конструктор по умолчанию инициализирует все поля нулевыми значениями. Также невозможно обойти конструктор по умолчанию, определяя начальные значения полей. Следующий код вызовет ошибку компиляции (Рисунок 1.3):

```
struct Dimensions
{
    public double Length = 3;
    public double Width = 4;
}
```

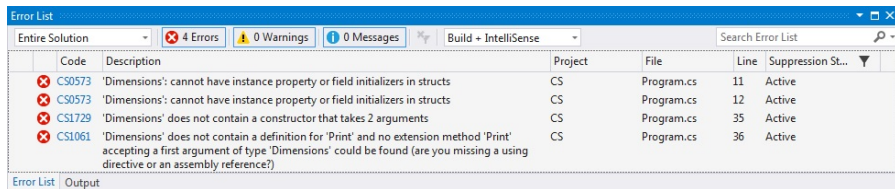



Рисунок 1.3. Ошибка при инициализации полей структуры

2. Синтаксис объявления класса

Классы C# — это некие шаблоны, по которым Вы можете создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные, и какую функциональность может иметь каждый конкретный объект (иногда называемый экземпляром) этого класса. Например, если у Вас есть класс, представляющий студента, он может определять такие поля, как `_studentID`, `_firstName`, `_lastName`, `_group`, и т.д. которые нужны ему для хранения информации о конкретном студенте.

Класс также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного студента, устанавливаете значения полей экземпляра и используете его функциональность. При создании классов, как и всех ссылочных типов — используется ключевое слово `new` для выделения памяти под экземпляр. В результате объект создается и инициализируется (помним, что числовые поля инициализируются нулями, логические — `false`, ссылочные — в `null`).

Синтаксис объявления и инициализации класса:

```
[модификаторы] Class имя_класса
{
    [модификаторы] тип_данных имя_поля;
```

```

[модификаторы] тип_данных имя_поля;
...
[модификаторы] тип_данных имя_метода ( [параметры] )
{
    //тело метода
}
...
}

```

Пример объявления класса, описывающего студента (Рисунок 2.1):

```

class Student
{
    int _studentID;
    string _firstName = "John";
    string _lastName = "Doe";
    string _group;

    public void Print()
    {
        Console.WriteLine($"Студент { _firstName }
                           { _lastName }");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student st1;
        st1 = new Student();
        st1.Print();

        Student st2 = new Student();
        st2.Print();
    }
}

```

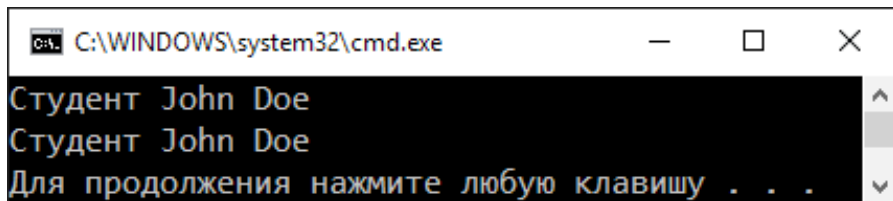


Рисунок 2.1. Пример объявления класса

Доступ к полям и методам класса осуществляется через оператор «.», который вызывается у объекта класса. Также следует напомнить, что доступ к содержимому класса вне его границ, мы имеем только к общедоступным данным. Остальные остаются инкапсулированными. С модификаторами мы познакомимся далее.

3. Модификаторы доступа языка программирования C#

Все типы и члены типов имеют уровень доступности, который определяет возможность их использования из другого кода в сборке разработчика или других сборках.

При определении класса его можно сделать открытым (`public`) или внутренним (`internal`). Открытый тип доступен любому коду любой сборки. Внутренний класс доступен только из сборки, где он определен. По умолчанию компилятор C# делает класс внутренним.

При определении члена класса (в том числе вложенного) можно указать модификатор доступа к члену. Модификаторы определяют, на какие члены можно ссылаться из кода. В CLR определен свой набор возможных модификаторов доступа, но в каждом языке программирования существует свой синтаксис и термины. Рассмотрим модификаторы определяющие уровень ограничения — от максимального (`private`) до минимального (`public`):

- **private** — данные доступны только методам внутри класса и вложенным в него классам;
- **protected** — данные доступны только методам внутри класса (и вложенным в него классам) или в любом из его дочерних классов;
- **internal** — данные доступны только в методах текущей сборки;

- **protected internal** — данные доступны только методам вложенного или производного типа класса и любым методам текущей сборки;
- **public** — данные доступны всем методам во всех сборках.

Вы также должны понимать, что доступ к члену класса можно получить, только если он определен в видимом классе. То есть, если в сборке А определен внутренний класс, имеющий открытый метод, то код сборки Б не сможет вызвать открытый метод, поскольку внутренний класс сборки А не доступен из Б.

В процессе компиляции кода компилятор проверяет корректность обращения кода к классам и членам. Обнаружив некорректную ссылку на какие-либо классы или члены, выдается ошибка компиляции.

Если не указать явно модификатор доступа, компилятор C# выберет по умолчанию **private** для членов класса и **internal** для самого класса.

Если в производном классе переопределяется член базового — компилятор C# потребует, чтобы у членов базового и производного классов был одинаковый модификатор доступа. При наследовании базовому классу CLR позволяет понижать, но не повышать уровень доступа к члену.

4. Поля класса

Любой класс может включать в себя множество данных. К таким данным относятся: поля, конструкторы класса, методы, перегруженные операторы, свойства, события, а также вложенные классы. Остановимся для начала на полях.

Поле — это переменная, которая хранит значение любого стандартного типа или ссылку на ссылочный тип. При объявлении полей могут указываться следующие ключевые слова:

- **static** — используется для объявления статического поля, которое принадлежит классу, а не конкретному объекту, то есть является общим для всех экземпляров класса.
- **const** — используется для объявления постоянного поля, то есть значение данного поля не может быть изменено. Поле с модификатором `const` должно быть обязательно проинициализировано при объявлении поля класса. Константные поля являются неявно статическими, поэтому обращение к ним необходимо осуществлять только через имя типа.
- **readonly** — означает, что поле будет использоваться только для чтения, присвоение значений таким полям разрешается только в конструкторе либо сразу при объявлении.

CLR поддерживает изменяемые (`read/write`) и неизменяемые (`readonly`) поля. Большинство полей — изменяемые.

Это означает, что значение таких полей может многократно меняться во время исполнения кода. Такие поля Вы уже видели в предыдущем примере в классе `Student`. Неизменяемые поля более гибкие, чем константы, за счет того, что значение им можно задать динамически, во время выполнения программы и после этого значение меняться не будет. Важно понимать, что неизменность поля ссылочного типа означает неизменность ссылки, которую оно содержит, но только не объекта, на которую эта ссылка указывает. То есть перенаправить ссылку на другое место в памяти мы не можем, но изменить значение объекта, на который указывает ссылка — можем. Значения неизменяемых полей значимых типов — изменять не можем.

Рассмотрим пример:

```
class MyClass
{
    public readonly int var = 10;
    public readonly int[] myArr = { 1, 2, 3 };
}

class Program
{
    static void Main(string[] args)
    {
        MyClass obj = new MyClass();
        obj.var = 100;                //Ошибка
        obj.myArr = new int[10];      //Ошибка
        obj.myArr[0] = 11;            //Ошибки нет
    }
}
```

Если объявляется статическое поле, то оно принадлежит классу в целом, а не конкретному объекту. И соответственно

получить доступ к такому объекту можно только через имя класса, используя следующий синтаксис:

```
имя_класса.имя_поля
```

Статическое поле класса является общим для всех экземпляров этого класса. Например, пускай у нас есть класс `Bank`. В этом классе будет статическое поле `balance`. Сымитируем ситуацию, когда в любом филиале банка можно будет положить деньги на депозит или взять кредит. Пусть все филиалы работают с общим счетом.

```
class Bank
{
    public static float balance = 1000000;
}
class Program
{
    static void Main(string[] args)
    {
        Bank filial1 = new Bank();
        Bank filial2 = new Bank();

        Console.WriteLine("Первому филиалу
                           доступно {0:C}",
                           Bank.balance);
        Console.WriteLine("Второму филиалу
                           доступно {0:C}",
                           Bank.balance);
        Console.WriteLine("В первом филиале
                           взяли кредит на
                           100000" + ", осталось {0:C}",
                           Bank.balance-=100000);
        Console.WriteLine("Второму филиалу
                           доступно {0:C}",
                           Bank.balance);
    }
}
```

```

    Console.WriteLine("В втором филиале
                        взяли кредит на
                        200000" + ", осталось {0:C}",
                        Bank.balance -= 200000);
    Console.WriteLine("Первому филиалу
                        доступно {0:C}",
                        Bank.balance);
    Console.WriteLine("В первом филиале
                        открыли депозит
                        на " + "200000, осталось
                        {0:C}",
                        Bank.balance += 200000);
    Console.WriteLine("Второму филиалу
                        доступно {0:C}",
                        Bank.balance);
}
}

```

Результаты выполнения программы (Рисунок 4.1):

```

C:\WINDOWS\system32\cmd.exe
Первому филиалу доступно 1 000 000,00 ?
Второму филиалу доступно 1 000 000,00 ?
В первом филиале взяли кредит на 100000, осталось 900 000,00 ?
Второму филиалу доступно 900 000,00 ?
В втором филиале взяли кредит на 200000, осталось 700 000,00 ?
Первому филиалу доступно 700 000,00 ?
В первом филиале открыли депозит на 200000, осталось 900 000,00 ?
Второму филиалу доступно 900 000,00 ?
Для продолжения нажмите любую клавишу . . .

```

Рисунок 4.1. Пример использования
статического поля класса

5. Конструкторы

Понятие конструктора

Конструктор — это специальный метод класса, который вызывается неявно при создании объекта с использованием ключевого слова `new`.

В C# существует три типа конструкторов. Несмотря на такое разнообразие, идея каждого — выполнить некоторые действия при создании объектов. Рассмотрим их по порядку.

Конструктор по умолчанию — не принимает никаких параметров и предоставляется компилятором при создании класса. Этот конструктор полезен тем, что он обнуляет все числовые типы, устанавливает в `false` логический, и в `null` — ссылочный, следовательно, все поля объекта будут проинициализированы значениями по умолчанию. В случае необходимости Вы можете переопределить конструктор по умолчанию под Ваши нужды.

Конструктор с параметрами — конструктор, который может принимать необходимое количество параметров для инициализации полей класса конкретными значениями.

Статический конструктор — конструктор, относящийся к классу, а не к объекту. Используется для инициализации статических полей класса. Определяется без какого-либо модификатора доступа с ключевым словом `static`.

При создании конструкторов нужно помнить, что все конструкторы (кроме статического) должны иметь модификатор доступа `public`, имя, совпадающее с именем класса,

и ничего не возвращать (даже `void`), также все, кроме конструктора по умолчанию и статического конструктора, могут иметь необходимое количество параметров. Конструктор по умолчанию может быть только один.

Еще один важный момент: если вы взялись за определение любого конструктора, то конструктор по умолчанию, который предоставлялся Вам компилятором, работать не будет!!! Компилятор решает, что теперь Вы несете ответственность за создание объектов класса и в его «услугах» не нуждается, поэтому он не будет создавать конструктор по умолчанию.

Рассмотрим пример определения своего конструктора по умолчанию, а остальные будут рассмотрены в следующих разделах. Для работы с конструкторами создадим новый класс описывающий машину.

```
class Car
{
    private string _driverName; //Имя водителя
    private int _currSpeed = 10; //Текущая скорость

    public Car()                //Конструктор по
                                //умолчанию

    {
        driverName = "Михаель Шумахер";
    }

    public void PrintState()     //Распечатка текущих
                                //данных

    {
        Console.WriteLine($"{_driverName}
                               едет со скоростью
                               {_currSpeed} км/ч.");
    }
}
```

```

public void SpeedUp(int delta) //Увеличение
                               //скорости
{
    _currSpeed += delta;
}

class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();
        for (int i = 0; i <= 10; i++)
        {
            myCar.SpeedUp(5);
            myCar.PrintState();
        }
    }
}

```

Результат выполнения программы (Рисунок 5.1):

```

C:\WINDOWS\system32\cmd.exe
Михаель Шумахер едет со скоростью 15 км/ч.
Михаель Шумахер едет со скоростью 20 км/ч.
Михаель Шумахер едет со скоростью 25 км/ч.
Михаель Шумахер едет со скоростью 30 км/ч.
Михаель Шумахер едет со скоростью 35 км/ч.
Михаель Шумахер едет со скоростью 40 км/ч.
Михаель Шумахер едет со скоростью 45 км/ч.
Михаель Шумахер едет со скоростью 50 км/ч.
Михаель Шумахер едет со скоростью 55 км/ч.
Михаель Шумахер едет со скоростью 60 км/ч.
Михаель Шумахер едет со скоростью 65 км/ч.
Для продолжения нажмите любую клавишу . . .

```

Рисунок 5.1. Использование конструктора по умолчанию

Параметризованный конструктор

Как было сказано выше конструктор с параметрами отличается от конструктора по умолчанию собственно наличием параметров. Поэтому сразу перейдем к примеру и добавим в наш класс `Car` параметризованный конструктор. Для экономии места старые поля и методы в примере отсутствуют, но в полной версии — есть.

```
class Car
{
    //Старые поля и методы...

    public Car(string name)
    {
        _driverName = name;
        _currSpeed = 10;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car("Рубенс Барикелло");
        for (int i = 0; i <= 10; i++)
        {
            myCar.SpeedUp(5);
            myCar.PrintState();
        }
    }
}
```

Результат выполнения кода (Рисунок 5.2):

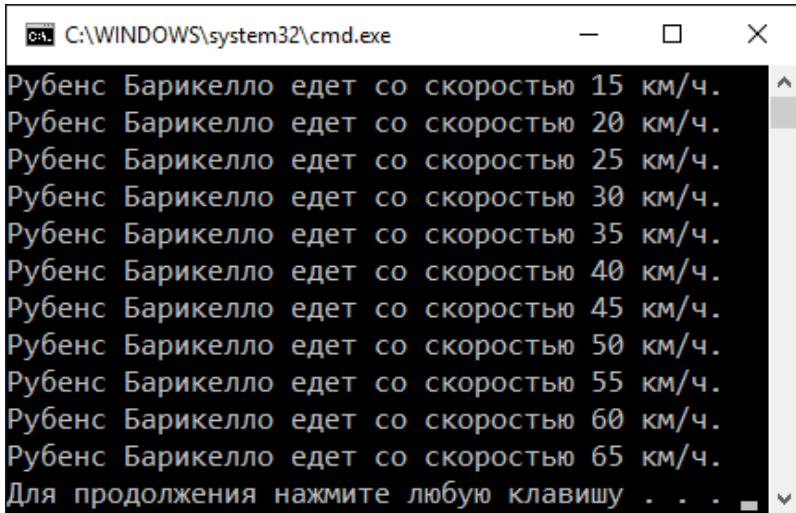


Рисунок 5.2. Использование конструктора с параметрами

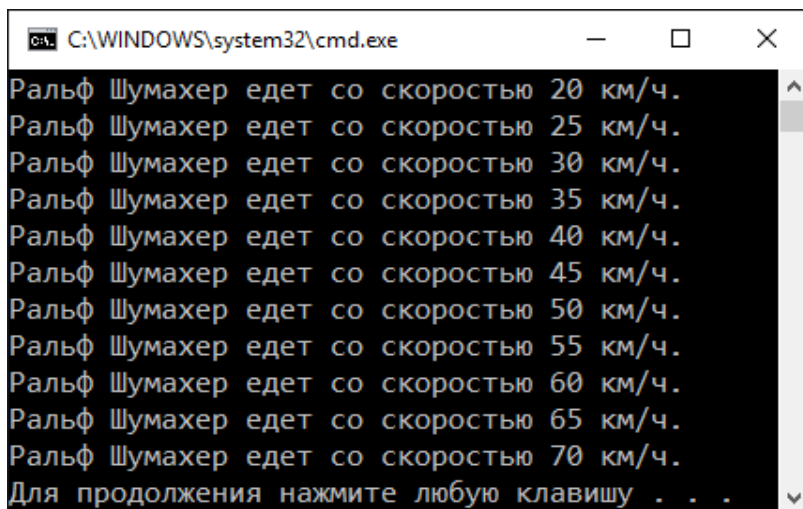
Перегруженные конструкторы

Поскольку конструкторы — это методы, а, как мы уже знаем, методы можно перегружать, то, следовательно, и конструкторов может быть сколько угодно. Единственное ограничение — конструктор по умолчанию должен быть один. Ну а количество параметризованных ограничено только лишь здравым смыслом. Вообще старайтесь определять только те конструкторы, с помощью которых было бы удобнее создавать объект. Вернемся к классу машины и определим еще один конструктор.

```
class Car
{
    //Старые поля и методы...
    public Car(string name, int speed)
    {
```

```
        _driverName = name;
        _currSpeed = speed;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car("Ральф Шумахер", 15);
        for (int i = 0; i <= 10; i++)
        {
            myCar.SpeedUp(5);
            myCar.PrintState();
        }
    }
}
```

Результат работы программы (Рисунок 5.3):



```
C:\WINDOWS\system32\cmd.exe
Ральф Шумахер едет со скоростью 20 км/ч.
Ральф Шумахер едет со скоростью 25 км/ч.
Ральф Шумахер едет со скоростью 30 км/ч.
Ральф Шумахер едет со скоростью 35 км/ч.
Ральф Шумахер едет со скоростью 40 км/ч.
Ральф Шумахер едет со скоростью 45 км/ч.
Ральф Шумахер едет со скоростью 50 км/ч.
Ральф Шумахер едет со скоростью 55 км/ч.
Ральф Шумахер едет со скоростью 60 км/ч.
Ральф Шумахер едет со скоростью 65 км/ч.
Ральф Шумахер едет со скоростью 70 км/ч.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 5.3. Использование перегруженного конструктора

Статический конструктор

Статический конструктор связан с классом, а не с конкретным объектом и нужен для инициализации статических данных. У статического конструктора существует целый ряд особенностей:

- в классе может быть только один статический конструктор;
- при объявлении статического конструктора нельзя передавать параметры и нельзя указывать модификаторы доступа;
- статический конструктор выполняется только один раз, независимо от количества созданных объектов данного класса;
- статический конструктор будет вызван до любого первого обращения к данному классу (обычно перед первым вызовом любого члена класса);
- статический конструктор никогда не вызывается никаким другим кодом, а только исполняющей средой .NET при загрузке класса;
- статический конструктор имеет доступ только к статическим членам класса.

Для примера со статическим конструктором создадим класс, описывающий банковские филиалы, но на этот раз статическое поле будет содержать бонус в процентах для оформления депозитов. А текущий баланс у каждого филиала будет свой.

```
class Bank
{
    private double _currBalance;
```

```

private static double _bonus;

public Bank(double balance)
{
    _currBalance = balance;
}
static Bank()
{
    _bonus = 1.04;
}
public static void SetBonus(double newRate)
{
    _bonus = newRate;
}
public static double GetBonus()
{
    return _bonus;
}
public double GetPercents(double summa)
{
    if ((_currBalance - summa) > 0)
    {
        double percent = summa * _bonus;
        _currBalance -= percent;
        return percent;
    }
    return -1;
}
}

class Program
{
    static void Main(string[] args)
    {
        Bank b1 = new Bank(1000000);
        Console.WriteLine("Текущий бонусный процент:
                           " + Bank.GetBonus());
    }
}

```

```
Console.WriteLine("Ваш депозит на {0:C},  
в кассе забрать: {1:C}",  
10000, b1.GetPercents(10000));  
}  
}
```

Результат выполнения программы (Рисунок 5.4):

Рисунок 5.4. Применение статического конструктора

6. Ключевое слово `this`

Ключевое слово `this` — это неявно присутствующая в классе ссылка на текущий экземпляр класса. Перечислим несколько типичных случаев использования `this`:

1. Одним из возможных вариантов применения `this` является необходимость устранения конфликта между именами параметров метода и именами полей класса:

```
class Student
{
    string firstName;
    public Student(string firstName)
    {
        this.firstName = firstName;
    }
}
```

2. Другое применение `this` — попытка избежать избыточности при инициализации членов класса. Как правило, в классе определяется главный конструктор (главным выбирают конструктор с максимальным количеством параметров), который содержит код инициализации, а все остальные конструкторы вызывают его, используя `this` с необходимыми параметрами, чтобы избежать дублирования кода. Рассмотрим снова пример с классом `Car`, внося в него следующие изменения:

```
class Car
{
    private string _driverName;
```

```

private int _currSpeed;
public Car():this("Нет водителя", 0){}
public Car(string driverName):this(driverName, 0){}
//Главный конструктор
public Car(string driverName, int speed)
{
    _driverName = driverName;
    _currSpeed = 10;
}
public void SetDriver(string driverName)
{
    _driverName = driverName;
}
public void PrintState()//Распечатка текущих данных
{
    Console.WriteLine($"{_driverName} едет
                        со скоростью
                        {_currSpeed} км/ч.");
}
//Увеличение скорости
public void SpeedUp(int delta)
{
    _currSpeed += delta;
}
}

class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();
        for (int i = 0; i <= 10; i++)
        {
            myCar.SpeedUp(5);
            myCar.PrintState();
        }
    }
}

```

Результат работы программы (Рисунок 6.1):

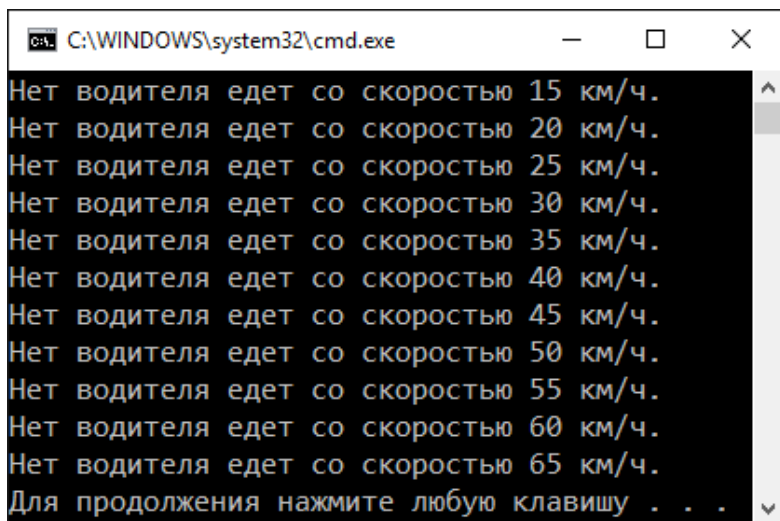


Рисунок 6.1. Использование ключевого слова `this`

- Еще один случай применения ключевого слова `this` — передача методу в качестве параметра ссылки на текущий объект:

```
public class ClassA
{
    public void MethodA(ClassB obj)
    {
        obj.MethodB(this);
    }
}

public class ClassB
{
    public void MethodB(ClassA obj)
    {
```

```

        Console.WriteLine("Работа с классом " +
                           obj.GetType().Name);
    }
}

public class Program
{
    public static void Main()
    {
        ClassA a = new ClassA();
        ClassB b = new ClassB();
        a.MethodA(b);
    }
}

```

Результат выполнения кода (Рисунок 6.2):

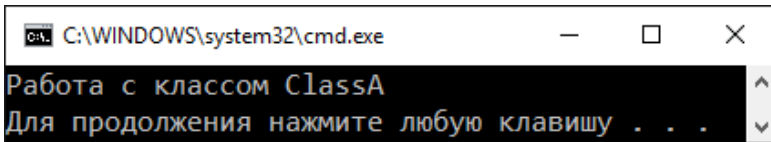


Рисунок 6.2. Использование ключевого слова `this` в методах

4. В последнем случае `this` используется для объявления индексаторов, работа с которыми будет обсуждаться в последующих уроках.

Применяя ключевое слово `this`, следует знать, что его использование в статических методах является недопустимым, так как они существуют только на уровне класса и не являются частями объектов этого класса.

7. Методы класса

Метод — это блок кода, содержащий ряд инструкций и расположенный в классе либо структуре. Методов вне определения классов — функций — в С# не существует.

Определение метода состоит из модификаторов, типа возвращаемого значения, за которым следует имя метода, затем — в круглых скобках список аргументов (если они есть) и далее — тело метода, заключенное в фигурные скобки:

```
[модификаторы] тип_возврата имя_метода ( [параметры] )  
{  
    //тело метода  
}
```

Добавим в класс `Student` метод. При этом сделаем все поля класса закрытыми. Используйте практику сокрытия данных класса и предоставления открытых методов по работе с этими данными. Тем самым Вы поддерживаете принцип инкапсуляции данных и уменьшаете их уязвимость.

```
class Student  
{  
    private string _firstName = "Петя";  
  
    public void ShowName()  
    {  
        Console.WriteLine(_firstName);  
    }  
}
```



```
class Program
{
    static void Main(string[] args)
    {
        Student st = new Student();
        st.ShowName();
    }
}
```

Результат выполнения кода (Рисунок 7.1):

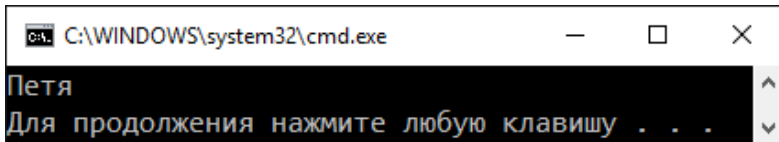


Рисунок 7.1. Пример работы с методами класса

Специально для работы со статическими полями — были введены статические методы. Эти методы, как и статические поля, принадлежат классу, а не объекту. Они исключают возможность вызова через объект класса и соответственно не работают с нестатическими полями.

Предположим, что все создаваемые нами студенты будут студентами академии «ШАГ» и соответственно добавим в наш класс `Student` статическое поле, которое будет содержать имя учебного заведения. Чтобы поле нельзя было изменить — сделаем его закрытым и позволим статическому методу `ShowAcademy` работать с нашим полем в режиме чтения.

```
class Student
{
    private static string _academyName="Компьютерная
                                академия \\"ШАГ\\"";
```

```

//старые поля и методы остаются без изменения

public static void ShowAcademy ()
{
    Console.WriteLine(_academyName);
}

class Program
{
    static void Main(string[] args)
    {
        Student.ShowAcademy();
    }
}

```

Результат выполнения программы (Рисунок 7.2):

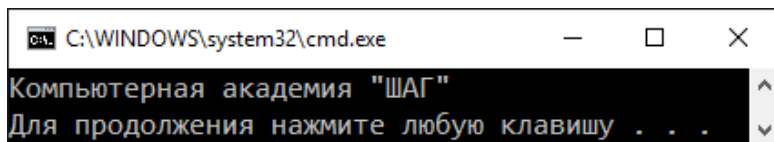


Рисунок 7.2. Пример работы статических методов

Передача параметров

Аргументы могут передаваться методу либо по значению, либо по ссылке. Когда переменная передается по ссылке, вызываемый метод работает с самой переменной, поэтому любые изменения, которые производятся над переменной внутри метода, останутся в силе после его завершения. С другой стороны, если переменная передается по значению, то вызываемый метод создает копию этой переменной, и соответственно все изменения в копии по

завершении метода будут утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

В C# все параметры передаются по значению, если Вы не укажете обратное. Однако нужно быть осторожным с пониманием этого в отношении ссылочных типов. Поскольку переменные ссылочного типа содержат лишь ссылку на объект, то именно ссылка будет скопирована при передаче параметра, а не сам объект, поэтому произведенные изменения сохранятся в самом объекте. В отличие от этого переменные значимых типов действительно содержат данные, поэтому в методе используются копии самих данных, что не приводит к изменению исходных значений. Рассмотрим пример вышесказанного:

```
class Program
{
    static void MyFunction(int[] MyArr, int i)
    {
        MyArr[0] = 100;
        i = 100;
    }

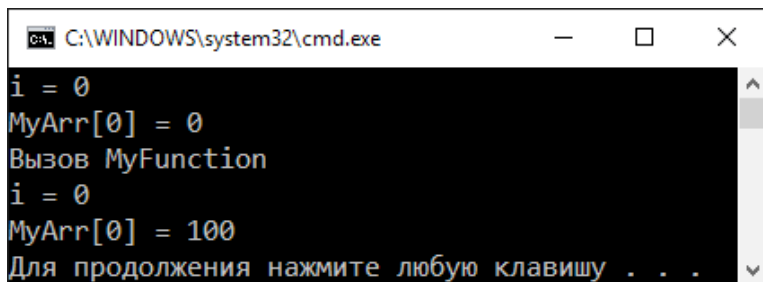
    static void Main(string[] args)
    {
        int i = 0;
        int[] MyArr = { 0, 1, 2, 4 };
        Console.WriteLine("i = " + i);
        Console.WriteLine("MyArr[0] = " + MyArr[0]);
        Console.WriteLine("Вызов MyFunction");
        MyFunction(MyArr, i);
    }
}
```

```

    Console.WriteLine("i = " + i);
    Console.WriteLine("MyArr[0] = " + MyArr[0]);
}
}

```

Результаты работы программы (Рисунок 7.3):



```

C:\WINDOWS\system32\cmd.exe
i = 0
MyArr[0] = 0
Вызов MyFunction
i = 0
MyArr[0] = 100
Для продолжения нажмите любую клавишу . . .

```

Рисунок 7.3. Пример работы с методом, принимающим параметры

Ключевое слово `return`

Помимо методов, которые не возвращают никакой информации, в C# существуют методы возвращающие данные. Синтаксис таких методов немного отличается: в заголовке метода добавляется тип возвращаемого значения, а само значение возвращается из метода с помощью ключевого слова `return`.

Метод может завершить свое выполнение тремя способами:

1. Когда управление дойдет до завершающей фигурной скобки (при этом метод ничего не возвращает).
2. Когда управление дойдет до ключевого слова `return` (без возвращаемого значения и тип возвращаемого значения метода — `void`).

3. Когда управление дойдет до ключевого слова `return` (после которого стоит возвращаемое значение, метод что-либо возвращает).

Добавим нашему классу `Student` метод, который будет возвращать полученную им оценку. Сама оценка будет генерироваться случайно (для чего будем использовать класс `Random`).

```
class Student
{
    //старые поля и методы остаются без изменения

    public int GetMark()
    {
        return new Random().Next(1, 12);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student st = new Student();
        Console.WriteLine("Оценка: " + st.GetMark());
    }
}
```

Возможный результат выполнения программы (Рисунок 7.4):

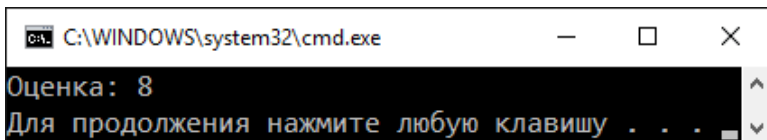


Рисунок 7.4. Применение ключевого слова `return`

Перегрузка методов

Перегрузка методов — определение нескольких методов с одинаковым именем и разной сигнатурой.

Еще раз уточним, что методы должны отличаться только сигнатурой (количеством, типами или порядком следования параметров). Тип значения, возвращаемого методом, так же как и модификаторы на перегрузку не влияют!

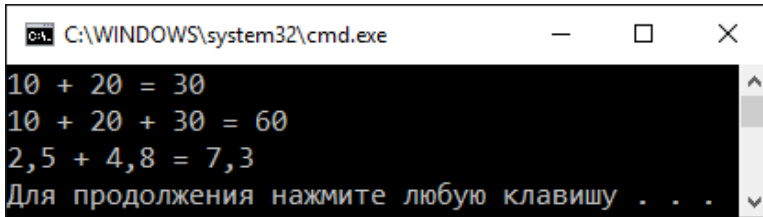
Рассмотрим пример, где создается класс `Mathematic`, в котором будет несколько перегруженных методов для сложения чисел.

```
class Mathematic
{
    public static int Sum(int a, int b)
    {
        return a + b;
    }
    public static int Sum(int a, int b, int c)
    {
        return a + b + c;
    }
    public static double Sum(double a, double b)
    {
        return a + b;
    }
}

class Program
{
    static void Main(string[] args)
    {
        int a = 10, b = 20, c = 30;
        double da = 2.5, db = 4.8;
```

```
Console.WriteLine($"{a} + {b} =  
    {Mathematic.Sum(a, b)}");  
Console.WriteLine($"{a} + {b} + {c} =  
    {Mathematic.Sum(a, b, c)}");  
Console.WriteLine($"{da} + {db} =  
    {Mathematic.Sum(da, db)}");  
}  
}
```

Результаты работы программы (Рисунок 7.5):



```
C:\WINDOWS\system32\cmd.exe  
10 + 20 = 30  
10 + 20 + 30 = 60  
2,5 + 4,8 = 7,3  
Для продолжения нажмите любую клавишу . . .
```

Рисунок 7.5. Пример перегрузки методов

8. Использование ref и out параметров

Как уже было сказано выше, параметры передаются в методы по значению (даже, как ни удивительно, ссылочные типы). Для того чтобы добиться передачи параметров через ссылку существуют ключевые слова `ref` и `out`. Убедимся на примере передачи по значению.

```
class Program
{
    private static void MyFunction(int i, int[] myArr)
    {
        Console.WriteLine("Внутри функции MyFunction  
до изменения i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
            Console.Write(val + " ");
        Console.WriteLine("}");

        i = 100;
        myArr = new int[] { 3, 2, 1 };

        Console.WriteLine("Внутри функции MyFunction  
после изменения i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
            Console.Write(val + " ");
        Console.WriteLine("}");
    }

    static void Main(string[] args)
    {

```



```

int i = 10;
int[] myArr = { 1, 2, 3 };

Console.WriteLine("Внутри метода Main до
                  передачи в метод " +
                  "MyFunction i = " + i);
Console.Write("MyArr { ");
foreach (int val in myArr)
    Console.Write(val + " ");
Console.WriteLine("}");

MyFunction(i, myArr);

Console.WriteLine("Внутри метода Main после передачи
                  в метод " + "MyFunction i = " + i);
Console.Write("MyArr { ");
foreach (int val in myArr)
    Console.Write(val + " ");
Console.WriteLine("}");
}
}

```

По результатам выполнения видим, что даже передача массива происходит по значению и в методе Main исходный массив не изменился (Рисунок 8.1).

```

C:\WINDOWS\system32\cmd.exe
Внутри метода Main до передачи в метод MyFunction i = 10
MyArr { 1 2 3 }
Внутри функции MyFunction до изменения i = 10
MyArr { 1 2 3 }
Внутри функции MyFunction после изменения i = 100
MyArr { 3 2 1 }
Внутри метода Main после передачи в метод MyFunction i = 10
MyArr { 1 2 3 }
Для продолжения нажмите любую клавишу . . .

```

Рисунок 8.1. Передача массива по значению

Использование модификатора `ref`

Ключевым словом `ref` помечаются те параметры, которые передаются в метод по ссылке. Таким образом, мы будем внутри текущего метода манипулировать данными, объявленными в вызывающем методе. Аргументы, которые передаются в метод с ключевым словом `ref`, обязательно должны быть проинициализированы, иначе компилятор выдаст сообщение об ошибке. Попробуем обозначить параметры из предыдущего примера как ссылочные и посмотрим, как изменится ситуация. Обратите внимание, что даже при вызове метода необходимо указывать ключевое слово `ref`, информирую о том, что аргументы передаются по ссылке.

```
class Program
{
    private static void MyFunction(ref int i, ref int[] myArr)
    {
        Console.WriteLine("Внутри функции MyFunction до  
изменения i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
            Console.Write(val + " ");
        Console.WriteLine("}");

        i = 100;
        myArr = new int[] { 3, 2, 1 };
        Console.WriteLine("Внутри функции MyFunction  
после изменения i = " + i);
        Console.Write("MyArr { ");
        foreach (int val in myArr)
            Console.Write(val + " ");
        Console.WriteLine("}");
    }
}
```

```

static void Main(string[] args)
{
    int i = 10;
    int[] myArr = { 1, 2, 3 };

    Console.WriteLine("Внутри метода Main
                      до передачи в метод " +
                      "MyFunction i = " + i);
    Console.Write("MyArr { ");
    foreach (int val in myArr)
        Console.Write(val + " ");
    Console.WriteLine("}");

    MyFunction(ref i, ref myArr);

    Console.WriteLine("Внутри метода Main после
                      передачи в метод " +
                      "MyFunction i = " + i);
    Console.Write("MyArr { ");
    foreach (int val in myArr)
        Console.Write(val + " ");
    Console.WriteLine("}");
}
}

```

Результаты выполнения программы изменились (Рисунок 8.2):

```

C:\WINDOWS\system32\cmd.exe
Внутри метода Main до передачи в метод MyFunction i = 10
MyArr { 1 2 3 }
Внутри функции MyFunction до изменения i = 10
MyArr { 1 2 3 }
Внутри функции MyFunction после изменения i = 100
MyArr { 3 2 1 }
Внутри метода Main после передачи в метод MyFunction i = 100
MyArr { 3 2 1 }
Для продолжения нажмите любую клавишу . . .

```

Рисунок 8.2. Использование модификатора ref

Использование модификатора out

Параметры, обозначенные ключевым словом `out`, также используются для передачи по ссылке. Отличие от `ref` состоит в том, что параметры считаются выходными и соответственно компилятор разрешает не инициализировать их до передачи в метод, но проследит, чтобы в методе этот параметр был **обязательно** проинициализирован (иначе будет выдано сообщение об ошибке — Рисунок 8.4).

```
class Program
{
    static void Main(string[] args)
    {
        int i;
        GetDigit(out i);
        Console.WriteLine("i = " + i);
    }
    private static void GetDigit(out int digit)
    {
        //return; // Error
        digit = new Random().Next(10);
    }
}
```

Возможный результат выполнения программы (Рисунок 8.3):

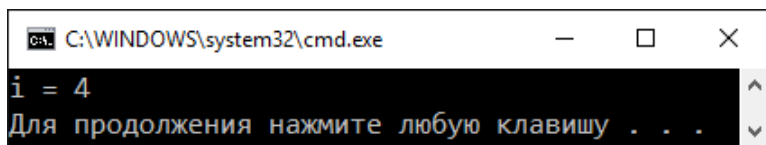
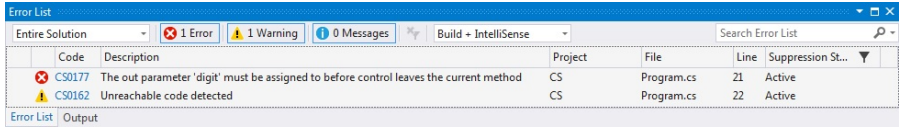


Рисунок 8.3. Использование модификатора out



Error List						
Entire Solution		1 Error	1 Warning	0 Messages	Build + IntelliSense	Search Error List
	Code	Description	Project	File	Line	Suppression St...
	CS0177	The out parameter 'digit' must be assigned to before control leaves the current method	CS	Program.cs	21	Active
	CS0162	Unreachable code detected	CS	Program.cs	22	Active

Error List Output

Рисунок 8.4. Ошибка: отсутствует инициализация параметра

9. Создание методов с переменным количеством аргументов

Для создания метода с переменным количеством аргументов можно было бы в качестве параметра передать массив значений. Рассмотрим пример такого метода.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Сумма = " + Sum(new int[] { 1,
            2, 3, 4, 5 }));
    }
    private static int Sum(int[] arr)
    {
        int res = 0;
        foreach (int i in arr)
            res += i;
        return res;
    }
}
```

Результат работы метода представлен на рисунке 9.1:

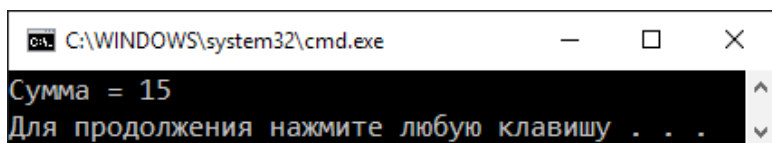


Рисунок 9.1. Работа метода с переменным количеством аргументов

Можем увидеть, что хотя результат правильный, но подход явно некрасивый. Поэтому специально для создания методов с переменным количеством аргументов существует ключевое слово `params`, которым помечают параметр метода. При использовании этого ключевого слова необходимо учитывать, что параметр, помечаемый ключевым словом `params`:

- должен стоять последним в списке параметров;
- должен указывать на одномерный массив любого типа.

Тот же пример, но с параметром типа `params`.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Сумма = " + Sum( 1, 2, 3, 4, 5 ));
    }
    private static int Sum(params int[] arr)
    {
        int res = 0;
        foreach (int i in arr)
            res += i;
        return res;
    }
}
```

В результате получаем тот же вывод (Рисунок 9.1), но код при этом стал более изящным.

В заключении следует отметить, что вызов метода с переменным количеством параметров отрицательно влияет на производительность, из-за постоянного размещения в куче элементов и их удаления. Поэтому, если это возможно, лучше определить перегруженные методы с различным количеством аргументов без использования `params`.

10. Частичные типы (partial types)

Частичные типы поддерживаются только компиляторами C# и некоторых других языков, но CLR ничего о них не знает.

Ключевое слово `partial` говорит компилятору C#, что исходный код класса может располагаться в нескольких файлах. Существуют две основные причины, по которым исходный код разбивается на несколько файлов.

1. **Управление версиями** — определение класса может содержать большое количество кода. Если этот класс будут одновременно редактировать несколько программистов, то по окончании работы им придется каким-то образом объединять свои результаты, что весьма неудобно.
2. **Разделители кода** — при создании в Microsoft Visual Studio нового проекта Windows Forms или Web Forms, некоторые файлы с исходным кодом создаются автоматически. Они называются шаблонными. При использовании конструкторов форм Visual Studio в процессе создания и редактирования элементов управления формы Visual Studio автоматически генерирует весь необходимый код и помещает его в отдельные файлы. Это значительно повышает продуктивность работы. Раньше автоматически генерируемый код помещался в тот же файл, где программист писал свой исходный

код. Проблема была в том, что при случайном изменении сгенерированного кода конструктор форм переставал корректно работать. Начиная с Visual Studio 2005, при создании нового проекта Visual Studio создает два исходных файла; один предназначен для программиста, а в другой помещается код, создаваемый редактором форм. Теперь вероятность случайного изменения генерируемого кода существенно меньше.

Ключевое слово `partial` применяется к типам во всех файлах с определением класса. При компиляции компилятор объединяет эти файлы, и готовый класс помещается в результирующий файл сборки с расширением `.exe` или `dll`. Как уже говорилось, частичные типы реализуются только компилятором C#, поэтому все файлы с исходным кодом типа необходимо писать на одном языке и компилировать их в единый модуль.

Например:

```
//Class1.cs
partial class MyClass
{
    public static void Method1 ()
    {
        Console.WriteLine("Класс: \"MyClass\" Метод:
                           \"Method1\"");
    }
}

//Class2.cs
partial class MyClass
{
    public static void Method2 ()
    {
```

```

        Console.WriteLine("Класс: \"MyClass\" Метод:
                           \"Method2\"");
    }
}

//Program.cs
class Program
{
    static void Main(string[] args)
    {
        MyClass.Method1();
        MyClass.Method2();
    }
}

```

Результат получается такой же, как будто класс объявлен в одном файле (Рисунок 10.1):

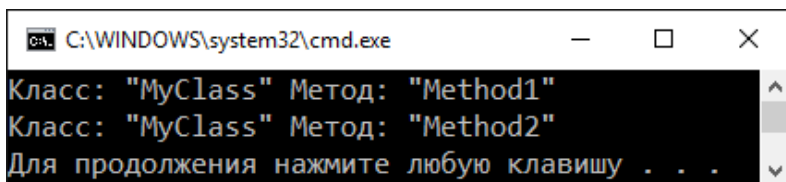


Рисунок 10.1. Использование частичных классов

11. Свойства

Если Вы прилежно изучали язык C++, тогда к этому моменту у Вас уже должен возникнуть вопрос: «А где же геттеры и сеттеры?» Данный раздел как раз и даст ответ на этот вопрос. Но прежде необходимо освежить знания тех, кто не совсем старательно учился или просто забыл.

Что такое свойства?

Одним из основных принципов объектно-ориентированного программирования является *инкапсуляция*, суть которой состоит в сокрытии реализации конкретного класса и предоставлении интерфейса для работы с ним. Сокрытие реализации обеспечивается путем назначения членам класса модификатора доступа `private`, а интерфейс взаимодействия в языке C++ обеспечивается определенными методами класса с модификаторами доступа `public`. Методы для получения какого-либо значения имеют общее название — геттеры, а методы для установления значения — сеттеры.

Подобная реализация на языке C# не вызовет ошибки и будет выглядеть следующим образом.

```
using static System.Console;

namespace SimpleProject
{
    class Example
    {
        int _num;
```

```

//метод для установления значения переменной num
public void Set(int num)
{
    _num = num;
}
//метод для считывания значения переменной num
public int Get()
{
    return _num;
}

class Program
{
    static void Main(string[] args)
    {
        Example example = new Example();
        Write("Введите целое число: ");
        example.Set(int.Parse(ReadLine()));
        Write("Вы ввели: ");
        WriteLine(example.Get());
    }
}

```

Возможный результат выполнения кода, приведенного выше (Рисунок 11.1).

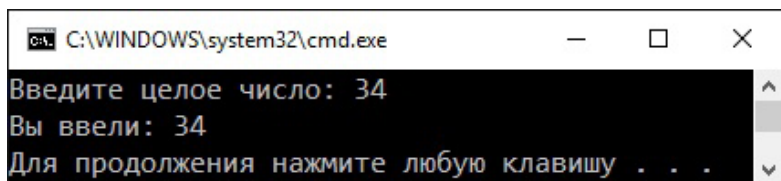


Рисунок 11.1. Использование методов для работы с полем класса

Помимо этого в языке C# предлагается более предпочтительный способ инкапсуляции данных — использование так называемых свойств (properties). Свойство является упрощенным представлением методов доступа и изменения, и обеспечивает как возможность присваивания значений приватным полям классов, независимо от их типа, так и считывание их значений, при необходимости реализовывая любую внутреннюю логику. То есть свойства интегрируют возможности двух разных по назначению групп методов.

Синтаксис объявления свойств

Синтаксис объявления свойства подобен синтаксису объявления метода, но без использования скобок. Обычно свойство имеет модификатор доступа `public`, хотя в некоторых случаях может быть `protected`, но не может быть `private`, иначе использование свойства утратит всякий смысл. Тип возврата свойства должен совпадать с типом поля, которое инкапсулировано в этом свойстве. Имя свойства, по договоренности, должно совпадать с именем инкапсулированного поля только начинаться с прописной буквы. Реализация свойства состоит из двух методов доступа или аксессоров `get` и `set`. Аксессор `get` позволяет получить значение инкапсулированного поля класса, а `set` обеспечивает изменение значения этого поля.

```
int _num;
public int Num
{
    get { return _num; }
    set { _num = value; }
}
```

Аксессор `get` обязательно должен включать оператор `return`, который возвращает поле класса.

В аксессоре `set` в качестве присваиваемого значения используется лексема `value`, которая является контекстно-зависимым ключевым словом. То есть соответствует любому значению, которое присваивается данному свойству и за пределами свойства оно может использоваться как обычный идентификатор, и при этом не будет возникать конфликта имён.

Перепишем предыдущий пример с использованием свойств, результат работы останется прежним (Рисунок 11.1).

```
using static System.Console;

namespace SimpleProject
{
    class Example
    {
        int _num;
        public int Num
        {
            get { return _num; }
            set { _num = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Example example = new Example();
            Write("Введите целое число: ");
            example.Num = int.Parse(ReadLine()); // set
        }
    }
}
```

```

        Write("Вы ввели: ");
        WriteLine(example.Num); // get
    }
}
}

```

Как Вы заметили, в коде явно не указывается, какой из аксессоров необходимо применить в данной ситуации — компилятор это определяет автоматически, основываясь на способе обращения к свойству в коде.

Существует упрощенный способ создания свойств. Для этого при создании свойства необходимо написать аббревиатуру `propfull` (Рисунок 11.2) и нажать два раза кнопку табуляции (Tab).

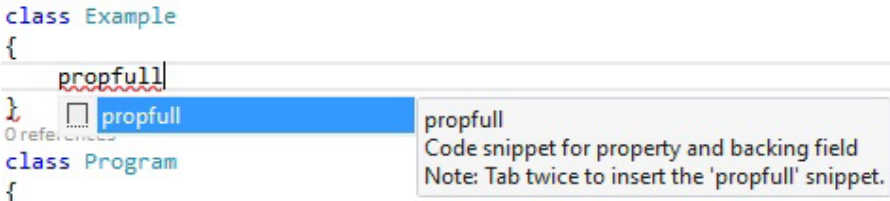


Рисунок 11.2. Упрощенный способ создания свойств (начало)

После этого будет генерирована заготовка свойства, в которой, при необходимости, можно самостоятельно задать необходимый тип данных поля класса, название поля и свойства, навигация между ними осуществляется при помощи кнопки табуляции (Рисунок 11.3).

```

class Example
{
    private int myVar;

```

```

0 references
public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}

```

Рисунок 11.3. Упрощенный способ создания свойств (продолжение)

Аксесоры `get` и `set` по сути являются методами, то есть могут иметь собственную логику. В частности в аксесоре `set` может быть размещён код проверки входных значений.

Например:

```

set
{
    if (value >= 0 && value <= 120)
    {
        _num = value;
    }
}

```

Здесь задаваемое значение проверяется на принадлежность диапазону от 0 до 120 включительно. Если всё благополучно, то новое значение будет присвоено полю `_num`, иначе его значение останется прежним.

Ещё один момент, на который следует обратить внимание — это модификатор доступа для каждого из аксесоров, как видите, он явно не указан. По умолчанию модификаторы доступа для обоих аксесоров соответствуют уровню доступа всего свойства в целом. Явно модификаторы доступа для аксесоров указываются только в том

случае, если они накладывают большие ограничения, чем модификатор самого свойства.

Например:

```
int _num;
public int Num
{
    get { return _num; }
    protected set { _num = value; }
}
```

Если Вы укажете для любого аксессуора модификатор доступа, совпадающий с модификатором свойства, то сгенерируется ошибка (Рисунок 11.4).

```
int _num;
public int Num
{
    get { return _num; }
    public set { _num = value; } // Error
}
```

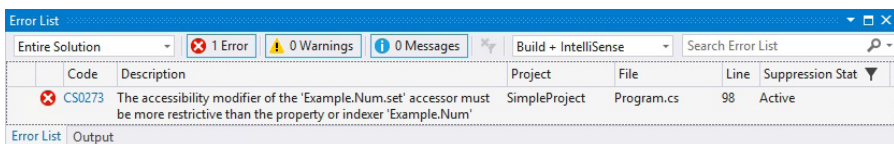


Рисунок 11.4. Ошибка: модификатор доступа у аксессуора должен быть более ограничительным, чем у свойства

Причём явно может быть указан модификатор только для одного из аксессуаров, модификатор доступа у второго аксессуора будет совпадать с модификатором свойства. Если модификатор доступа будет явно указан и для второго аксессуора, тогда сгенерируется ошибка (Рисунок 11.5).

```
int _num;
public int Num
{
    private get { return _num; } // Error
    protected set { _num = value; }
}
```

The screenshot shows the 'Error List' window in Visual Studio. The top bar indicates '1 Error' and '0 Warnings'. The error table contains one entry:

Code	Description	Project	File	Line	Suppression Stat
CS0274	Cannot specify accessibility modifiers for both accessors of the property or indexer 'Example.Num'	SimpleProject	Program.cs	94	Active

Below the table, the 'Error List' tab is selected, and the 'Output' window is visible.

Рисунок 11.5. Ошибка: невозможно указать модификаторы для обоих аксессоров

Существует возможность реализации только одного из аксессоров либо `get`, либо `set`, в этом случае получится свойство только для чтения либо только для записи соответственно. Например, свойство только для чтения будет выглядеть следующим образом.

```
int _num;
public int Num
{
    get { return _num; }
}
```

В этом случае изменять значения поля `_num` можно только в пределах его класса, а для получения значения этого поля нет никаких ограничений. Аналогичного результата можно добиться, если указать модификатор доступа `private` для аксессора `set`.

```
int _num;
public int Num
{
    get { return _num; }
}
```

Свойство может быть статическим. Это нужно в том случае, если свойство нужно для задания или считывания значения статического приватного поля класса. При этом их вызов осуществляется точно так же, как и у статических методов — через имя класса, возможный результат работы останется прежним (Рисунок 11.1).

```
using static System.Console;

namespace SimpleProject
{
    class Example
    {
        static int _num;
        public static int Num
        {
            get { return _num; }
            set { _num = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Write("Введите целое число: ");
            Example.Num = int.Parse(ReadLine()); // set
            Write("Вы ввели: ");
            WriteLine(Example.Num); // get
        }
    }
}
```

Нельзя разделять свойство на 2 части для каждого из аксессоров, т.е. следующая запись является недопустимой

и будет сгенерирована ошибка на этапе компиляции (Рисунок 11.6).

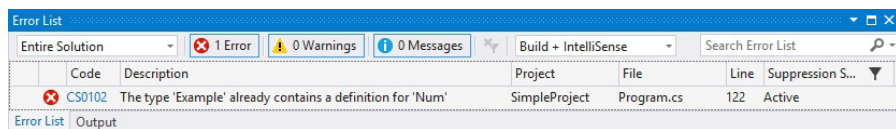


Рисунок 11.6. Ошибка: тип 'Example' уже содержит определение для свойства Num

Примеры использования свойств

Прежде чем будет представлен пример использования автоматических свойств, хотелось бы остановиться еще на одном механизме — синтаксисе инициализации объектов. Этот способ значительно упрощает создание проинициализированных объектов.

Инициализатор объектов прописывается при объявлении класса в виде инициализаций отдельных свойств, разделенных запятыми, которые помещены внутри фигурных скобок. При этом до инициализации свойств существует возможность вызова любого конструктора данного класса, что в принципе не обязательно, но надо иметь в виду, что инициализация свойств переопределит значения соответствующих полей заданных в конструкторе. Общая форма записи выглядит следующим образом.

```
new Имя_класса [ (вызов любого конструктора) ]
    { Имя_свойства = значение,
      Имя_свойства = значение }
```

Подобным образом можно инициализировать класс с общедоступными полями, но мы отбросим эту

возможность как недопустимую (вспомните про инкапсуляцию).

В следующем примере отображён класс `Employee` с четырьмя приватными полями: имя, фамилия, возраст и зарплата сотрудника. Для каждого из полей класса предусмотрено открытое свойство с двумя аксессорами. В свойствах осуществляется дополнительная проверка задаваемых значений. Имя и фамилия приводятся к верхнему регистру, возраст проверяется на принадлежность интервалу допустимых значений, зарплата не может быть отрицательной величиной. Конструкторы в этом классе не используются, инициализация полей класса осуществляется через свойства благодаря синтаксису инициализации объектов. Результат работы программы (Рисунок 11.7).

```
using static System.Console;

namespace SimpleProject
{
    class Employee
    {
        string _firstName;
        public string FirstName
        {
            get { return _firstName != null ?
                _firstName : "Не задано"; }
            set { _firstName = value.ToUpper(); }
        }

        string _lastName;
        public string LastName
        {
```

```

        get { return _lastName != null ?
            _lastName : "Не задано"; ; }
        set { _lastName = value.ToUpper(); }
    }

    int _age;
    public int Age
    {
        get { return _age; }
        set { _age = (value > 115 || value < 1)
            ? 0 : value; }
    }

    float _wage;
    public float Wage
    {
        get { return _wage; }
        set { _wage = value < 0 ? 0 : value; }
    }

    public string Print()
    {
        return $"Имя: {FirstName}\nФамилия:
            {LastName}\nВозраст: {Age}\nЗарплата:
            {Wage}\n";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Employee { FirstName =
            "Дмитрий", LastName =
            "Сикорский", Age = 19,
            Wage = 4800f };
        Employee emp2 = new Employee();
    }
}

```

```

emp2.FirstName = "Денис";
// свойство LastName не установлено
// попытка присвоить невозможный возраст
emp2.Age = 120;
//попытка задать зарплату со знаком минус
emp2.Wage = -1000;
Employee emp3 = new Employee { FirstName =
    "Наталья", LastName =
    "Борисова", Age = 29,
    Wage = 2500f };
WriteLine(emp1.Print());
WriteLine(emp2.Print());
WriteLine(emp3.Print());
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Имя: ДМИТРИЙ
Фамилия: СИКОРСКИЙ
Возраст: 19
Зарплата: 4800

Имя: ДЕНИС
Фамилия: Не задано
Возраст: 0
Зарплата: 0

Имя: НАТАЛЬЯ
Фамилия: БОРИСОВА
Возраст: 29
Зарплата: 2500

Для продолжения нажмите любую клавишу . . .

```

Рисунок 11.7. Пример использования свойств

Автоматические свойства

Хотя свойства могут содержать проверки присваиваемых значений и изменять вид возвращаемых значений, что было продемонстрировано в предыдущем примере. Однако это происходит далеко не всегда и довольно часто бывает ситуация когда свойства используются только для присваивания или возврата данных.

```
int _num;
public int Num
{
    get { return _num; }
    set { _num = value; }
}

string _firstName;
public string FirstName
{
    get { return _firstName; }
    set { _firstName = value; }
}
```

Именно в таких случаях рекомендуется использовать автоматические свойства.

Что такое автоматические свойства?

При объявлении автоматического свойства Вы пишете аксессоры без какой-либо реализации, и выглядит это следующим образом.

```
public int Num { get; set; }
public string FirstName { get; set; }
```


Особенность автоматического свойства заключается в следующем, компилятор на основании данного свойства автоматически создает некое скрытое поле с модификатором доступа `private` и осуществляет связь этого поля с написанным Вами свойством. В дальнейшем Вы осуществляете манипуляции только со свойством, а всю работу по взаимодействию со скрытым полем берет на себя компилятор.

Чтобы не писать код создания автоматического свойства вручную можно воспользоваться «услугами» Visual Studio уже известным Вам способом. На этот раз необходимо написать аббревиатуру `prop` и нажать два раза кнопку табуляции.

При использовании автоматических свойств запрещается создавать свойство без аксесора `get`, это приведет к ошибке на этапе компиляции (Рисунок 11.8).

```
public int Num { get; }
public string FirstName { set; } // Error
```

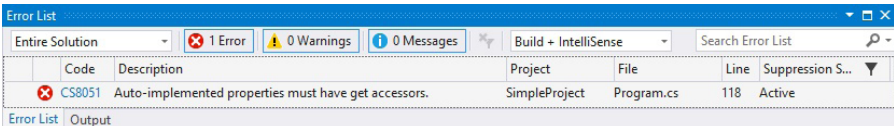


Рисунок 11.8. Ошибка: у автоматического свойства должен быть аксесор `get`

Применение модификаторов доступа в автоматических свойствах осуществляется, так же как и при использовании простых свойств. Приведем примеры создания автоматических свойств только для чтения и только для записи.

Заготовку автоматического свойства только для чтения тоже можно создать при помощи Visual Studio, так же

как было описано выше, только на этот раз необходимо написать аббревиатуру `prorg` и все также нажать два раза кнопку табуляции.

Инициализация автоматических свойств

При создании автоматического свойства, созданное компилятором скрытое поле инициализируется значением по умолчанию в зависимости от типа данных свойства: `int` — 0, `string`—`null`, `double` — 0.0 и т.д. Это отображено при помощи следующего кода.

```
using static System.Console;

namespace SimpleProject
{
    class Example
    {
        public int Num { get; set; }
        public string FirstName { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Example example = new Example();

            if (example.FirstName == null)
            {
                example.FirstName = "John";
            }
            WriteLine($"Имя: {example.FirstName} \
                nНомер:{example.Num}");
        }
    }
}
```

Результат работы программы (Рисунок 11.9).

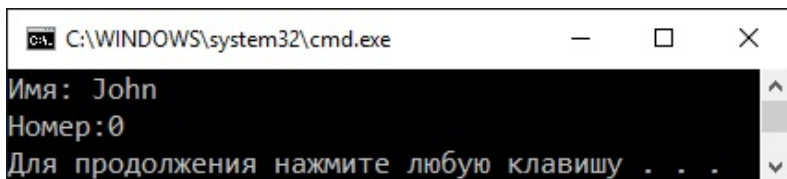


Рисунок 11.9. Значения по умолчанию
при работе со свойствами

В C# 6.0 появилась возможность инициализировать свойства начальными значениями при объявлении. Для этого необходимо написать присваивание этих значений после объявления самого свойства, при этом в качестве начального значения может быть любое выражение. Следующий код демонстрирует эту возможность — инициализация идентификатора в зависимости от значения текущего года.

```
using System;
using static System.Console;
namespace SimpleProject
{
    class Example
    {
        public int Id { get; } = DateTime.Now.Year <
            2000 ? 1001 : 2001;
        public int Num { get; set; } = 675;
        public string FirstName { get; set; } =
            "John";
    }
    class Program
    {
        static void Main(string[] args)
```

```

    {
        Example example = new Example();

        WriteLine($"Имя: {example.FirstName}\nНомер: {example.Num}\nId: {example.Id}");
    }
}

```

Результат работы программы (Рисунок 11.10).

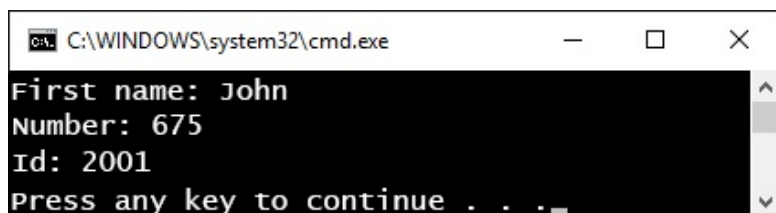


Рисунок 11.10. Инициализация свойств начальными значениями

Примеры использования автоматических свойств

В качестве примера возьмем класс `Student` и создадим автоматические свойства: имя, фамилия, дата рождения и название группы. Для имени и фамилии зададим значения по умолчанию.

```

using System;
using static System.Console;

namespace SimpleProject
{
    class Student

```

```

{
    public string FirstName { get; set; } = "John";
    public string LastName { get; set; } = "Doe";
    public string Group { get; set; }
    public DateTime DateBirth { get; set; }
    public string Print()
    {
        return $"Имя: {FirstName}\nФамилия:
                {LastName}\nДата рождения:
                {DateBirth.ToString()}\n
                nГруппа: {Group}\n";
    }
}

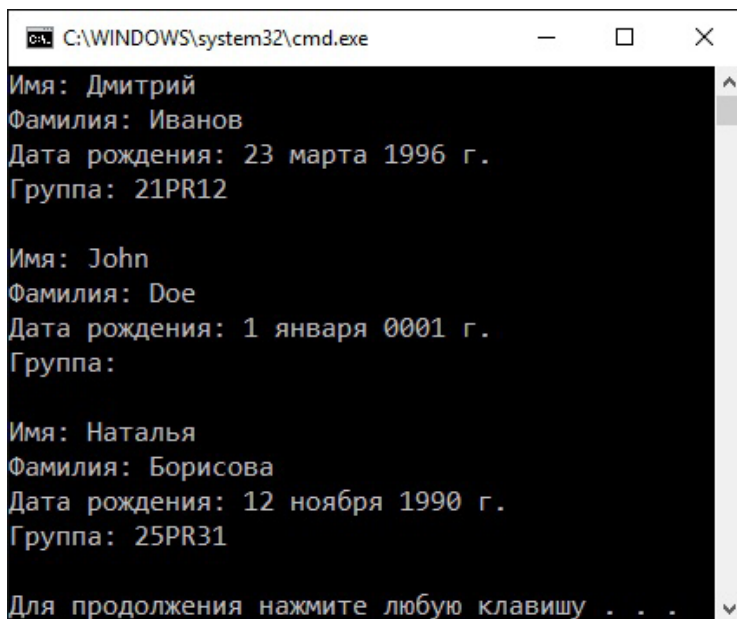
class Program
{
    static void Main(string[] args)
    {
        Student student1 = new Student { FirstName =
            "Дмитрий", LastName =
            "Иванов", DateBirth =
            new DateTime(1996,3,23),
            Group = "21PR12" };

        Student student2 = new Student();
        Student student3 = new Student { FirstName =
            "Наталья", LastName =
            "Борисова", DateBirth =
            new DateTime(1990,11,12),
            Group = "25PR31" };

        WriteLine(student1.Print());
        WriteLine(student2.Print());
        WriteLine(student3.Print());
    }
}

```

Результат работы программы (Рисунок 11.11).



```
C:\WINDOWS\system32\cmd.exe
Имя: Дмитрий
Фамилия: Иванов
Дата рождения: 23 марта 1996 г.
Группа: 21PR12

Имя: John
Фамилия: Doe
Дата рождения: 1 января 0001 г.
Группа:

Имя: Наталья
Фамилия: Борисова
Дата рождения: 12 ноября 1990 г.
Группа: 25PR31

Для продолжения нажмите любую клавишу . . .
```

Рисунок 11.11. Использование автоматических свойств

Null-conditional оператор

В версии C# 6.0, появился новый оператор проверки на `null` (null-conditional operator), при помощи которого цепочку проверки объектов на `null` значения можно написать в более сокращенном виде. Существуют две синтаксические формы null-conditional оператора: первая применяется при операциях с переменными nullable типа или объектами, и выглядит, как знак вопроса перед оператором «точка» (`?.`), вторая — как знак вопроса в сочетании с оператором `index` (`?[index]`) и применяется при работе с коллекциями объектов, доступных по индексу.

Для проверки значения на `null`, в предыдущих версиях языка C# использовался условный оператор.

```
Student student = null;
DateTime? date = null;
if (student != null)
{
    date = student.DateBirth;
}
```

Если переписать этот код, но с использованием оператора проверки на `null`, то мы получим тот же результат, но запись станет более лаконичной:

```
Student student = null;
DateTime? date = student?.DateBirth;
```

Если же необходимо проверить на `null` свойства класса, которые сами тоже являются классом, тогда мы могли получить большую условную конструкцию, которую можно заменить довольно объемным тернарным оператором.

```
Student student = new Student();
string groupName = student == null ? null : student.
Group == null ? null : student.Group.Name;
```

Использование `null`-условного оператора позволяет получить более упрощенную запись последовательной проверки значений на равенство `null`.

```
Student student = new Student();
string groupName = student?.Group?.Name;
```

Следующий код демонстрирует все варианты использования `null`-условного оператора. В примере работы с массивом, оператор проверки на `null` дополнен еще и операцией `??` для создания экземпляра класса и инициализации свойства `Group`, если в результате получим `null`, иначе вызов метода `Print()` приведет к ошибке на этапе выполнения.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Group
    {
        public string Name { get; set; }
    }

    class Student
    {
        public string FirstName { get; set; } = "John";
        public string LastName { get; set; } = "Doe";
        public Group Group { get; set; }
        public DateTime DateBirth { get; set; }
        public string Print()
        {
            return $"Имя: {FirstName}\n" +
                   "Фамилия: {LastName}\n" +
                   "Дата рождения: {DateBirth}.\n" +
                   "Группа: {Group.Name}\n";
        }
    }

    class Program
    {
```



```

static void Main(string[] args)
{
    WriteLine («Экземпляр класса Student
               не создан.»);
    Student student1 = null;
    DateTime? date = student1?.DateBirth;
    WriteLine ($"\n\tДата рождения: {date}");

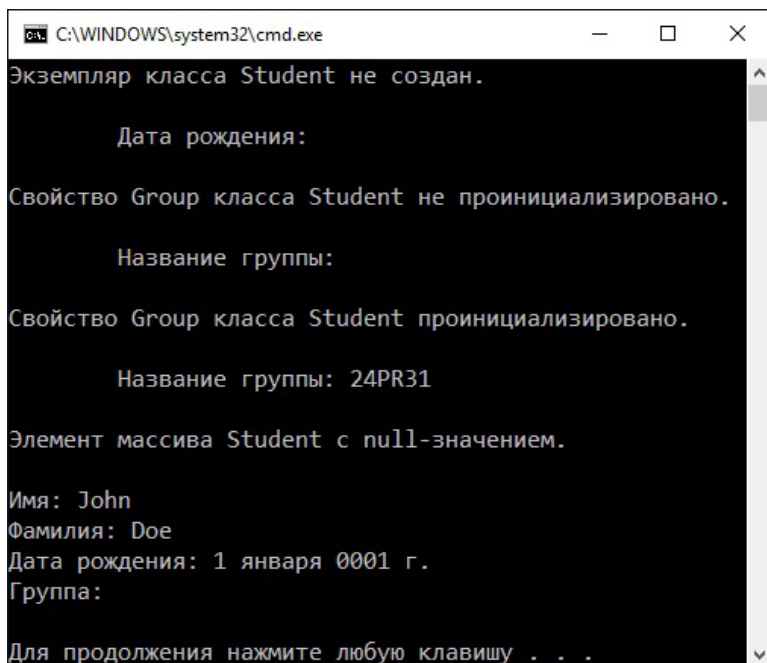
    WriteLine («\nСвойство Group класса Student
               не проинициализировано.»);
    Student student2 = new Student();
    string groupName = student2?.Group?.Name;
    WriteLine ($"\n\tНазвание группы:
               {groupName}");

    WriteLine («\nСвойство Group класса Student
               проинициализировано.»);
    Student student3 = new Student { Group =
                                     new Group { Name = "24PR31" } };
    groupName = student3?.Group?.Name;
    WriteLine ($"\n\tНазвание группы:
               {groupName}");

    WriteLine ("\nЭлемент массива Student
               с null-значением.\n");
    Student[] students = { student1,
                           student2, student3 };
    Student student = students?[0] ??
                       new Student { Group = new Group() };
    WriteLine(student.Print());
}
}

```

Результат работы программы (Рисунок 11.12).



```
C:\WINDOWS\system32\cmd.exe
Экземпляр класса Student не создан.

    Дата рождения:

Свойство Group класса Student не проинициализировано.

    Название группы:

Свойство Group класса Student проинициализировано.

    Название группы: 24PR31

Элемент массива Student с null-значением.

Имя: John
Фамилия: Doe
Дата рождения: 1 января 0001 г.
Группа:

Для продолжения нажмите любую клавишу . . .
```

Рисунок 11.12. Использование автоматических свойств

Многие из тех, кто дочитает до конца данного раздела, подумают: «Причем этот раздел к свойствам?» Да, данную тему можно было осветить и раньше, но тогда Вам были бы не понятны приведенные здесь примеры.

12. Пространство имён

Что такое пространство имён?

А теперь поближе познакомимся с одним из основополагающих понятий языка C#, без которого так или иначе не обходится ни одна программа. Речь пойдёт о так называемых пространствах имён. Что же представляет собой пространство имён в C#?

Пространство имён — это некая область объявления данных, обеспечивающая логическую взаимосвязь типов.

Другими словами, пространство имён — это способ группирования ассоциированных типов, включая классы, структуры, делегаты, интерфейсы, перечисления, а также другие пространства имён.

При создании класса в файле проекта Вы можете поместить его в пространство имён, которое сами ему назначите. Если позднее Вы решите включить в программу другой класс (возможно, уже в другом файле), выполняющий какую-то ассоциированную задачу, то поместите его в это же пространство имён, создавая логическую группу. Тем самым Вы указываете на то, как данные классы взаимодействуют друг с другом.

Если какой-нибудь тип (пусть это будет класс `Poetry`) был объявлен в своём пространстве имён (пусть оно называется `Inspiration`), то это обозначает, что этот тип будет мирно существовать в своём пространстве имён, не конфликтуя с одноимёнными типами, объявленными в пространствах имён, отличных от данного.

Честно говоря, понятие пространства имён очень жизненное.

Например, одного моего соседа зовут Алексей, а в школе у меня был одноклассник, которого тоже так звали. А в коллективе на работе есть один товарищ, которого (Вы не поверите!!!) тоже зовут Алексей.

Список можно продолжать...

И хотя всех этих людей объединяет одно имя, их не спутать, потому что каждый из них ограничен логикой своего контекста. Для одного это класс в школе, для другого — дом, где я живу, а для третьего — коллектив, в котором сейчас работаю.

Переводя это в синтаксис языка C#, информацию об этих людях можно отобразить следующим образом:

- ШкольныйКласс.Алексей
- МойДом.Алексей
- КоллективНаРаботе.Алексей

Программирование не отстаёт от жизни, и поэтому очень похожая ситуация существует и здесь. Представить себе невозможно, сколько библиотек, классов, функций, переменных и тому подобного богатства, обозначенного каждый своим идентификатором, уже существует на сегодняшний день. Не мудрено, что программисту, работающему над созданием новых библиотек, классов и тому подобного добра, легко дать своему детищу имя, которым уже когда-то наградили другой тип. Гигабайты человеческой памяти, к сожалению, не в состоянии удерживать такое количество именований (если тот, кто читает этот урок, может продемонстрировать обратное, заранее прошу прощения).

Та же ситуация просматривается в библиотеке .NET. В пространстве имён `System.Web.UI.Controls` есть класс `Button`, и в пространстве имён `System.Windows.Forms` таковой имеется. Однако каждый из них существует в рамках своего пространства имён, и конфликта не возникает.

Цели и задачи пространства имён

Итак, Вы уже, наверное, догадались, что одной из основных задач создания пространств имён является предотвращение конфликтов по совпадению имён. Рассмотрим следующий пример:

```
using System;

class Foo
{
    //...
}
class Foo
{
    //...
}
```

В ответ на такую запись компилятор выдаёт следующее сообщение об ошибке (Рисунок 12.1):

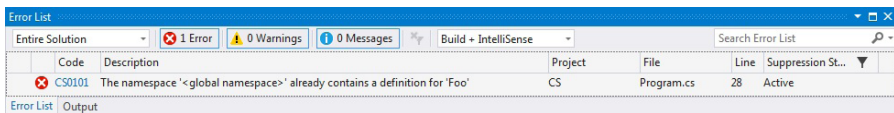


Рисунок 12.1. Ошибка: объявление классов с одинаковым именем

Оба класса объявлены в одном файле, соответственно они находятся в одной области видимости. А поскольку идентификаторы у них совпадают, возникает одна из самых распространённых ситуаций — конфликт имён.

Для решения этой проблемы нужно либо изменить название одного из классов, либо разграничить их области видимости с помощью пространств имён. В данной ситуации, конечно, можно обойтись и первым способом, примерно так:

```
using System;

class Foo
{
    //...
}
class Foo1
{
    //...
}
```

Но в другом случае это может быть критично. Тогда конфликт можно полностью исчерпать с помощью пространства имён:

```
using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
```

```

}
class Foo
{
    //...
}

```

либо так:

```

using System;

namespace CSharpNamespace
{
    class Foo
    {
        //...
    }
}
namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }
}

```

Ещё один пример:

```

using System;

namespace A
{
    class Incrementer
    {
        private int _count;

        public Incrementer(int count)
        {

```

```

        _count = count;
    }
    public int MultyIncrement()
    {
        for (int i = 0; i < 5; i++)
            _count++;
        return _count;
    }
}

namespace B
{
    class Incrementer
    {
        private int _var;

        public Incrementer(int var)
        {
            _var = var;
        }

        public int AnotherMultyIncrement()
        {
            for (int i = 0; i < 5; i++)
                _var += 10;
            return _var;
        }
    }

    class Tester
    {
        public static void Main()
        {
            Incrementer obj1 =
                new Incrementer(10);
            Console.WriteLine(obj1.
                AnotherMultyIncrement());
        }
    }
}

```



```

        A.Incrementer obj2 =
            new A.Incrementer(5);
        Console.WriteLine(obj2.
            MultyIncrement());
    }
}

```

Результат работы программы (Рисунок 12.2):

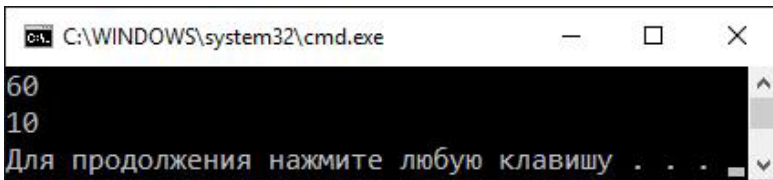


Рисунок 12.2. Использование одноименных классов

В данной программе в одном файле в двух разных пространствах имён А и В размещены два класса с одинаковым именем `Incrementer`. Точка входа программы — метод `Main` размещён во втором из классов. В `Main` создаётся экземпляр класса `Incrementer` из пространства имён В, и для него вызывается метод, увеличивающий значение поля

```

Incrementer obj1 = new Incrementer(10);
Console.WriteLine(obj1.AnotherMultyIncrement());

```

Затем создаётся ещё один объект — экземпляр класса `Incrementer` из пространства имён А.

```

A.Incrementer ob2 = new A.Incrementer(5);
Console.WriteLine(ob2.MultyIncrement());

```

Обратим внимание на запись: здесь имя класса указывается в сочетании с названием пространства имён.

Почему так? Пространство создаёт область видимости для всех объектов (читай типов), находящихся в его пределах. Внутри пространства имён А не видны объекты из пространства имён В и наоборот. Для того чтобы в пространстве имён В обратиться к типу из пространства имён А, необходимо явно указать имя второго.

А что будет, если мы поступим иначе и не добавим название пространства имён А перед именем класса `Incrementer`? Поменяем запись и заново запустим программу.

Код теперь выглядит так:

```
class Tester
{
    public static void Main()
    {
        Incrementer obj1 = new Incrementer(10);
        Console.WriteLine(obj1.
                           AnotherMultyIncrement());
        Incrementer obj2 = new Incrementer(5);
        Console.WriteLine(obj2.MultyIncrement());
    }
}
```

А компилятор выдаёт сообщение об ошибке (Рисунок 12.3):

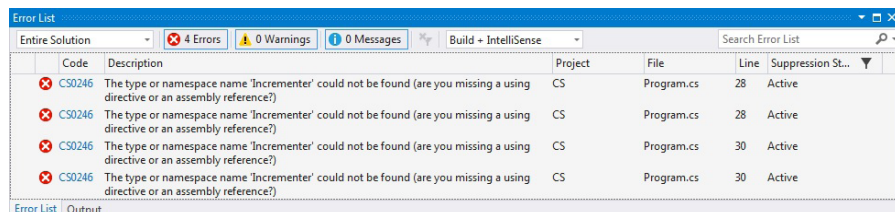


Рисунок 12.3. Ошибка на этапе компиляции

Это происходит потому, что компилятор воспринимает объявление второго объекта как попытку создать ещё один экземпляр класса `Incrementer` из пространства имён `B`. А поскольку в нём нет определения метода под названием `MultyIncrement`, генерируется ошибка.

Предположим, в обоих пространствах имён в соответствующих классах `Incrementer` определён метод с названием `MultyIncrement`. В методе `Main` создаётся два экземпляра класса `Incrementer` с последующим вызовом метода `MultyIncrement` для обоих. Всё это отображено в следующем коде:

```
using System;

namespace A
{
    public class Incrementer
    {
        private int _count;
        public Incrementer(int count)
        {
            _count = count;
        }

        public int MultyIncrement()
        {
            for (int i = 0; i < 5; i++)
                _count++;
            return _count;
        }
    }
}

namespace B
{
```

```

public class Incrementer
{
    private int _var;

    public Incrementer(int var)
    {
        _var = var;
    }

    public int MultyIncrement()
    {
        for (int i = 0; i < 5; i++)
            _var += 10;
        return _var;
    }
}

class Tester
{
    public static void Main()
    {
        Incrementer obj1 = new Incrementer(10);
        Console.WriteLine(obj1.MultyIncrement());
        Incrementer obj2 = new Incrementer(5);
        Console.WriteLine(obj2.MultyIncrement());
    }
}

```

Результат работы программы (Рисунок 12.4):

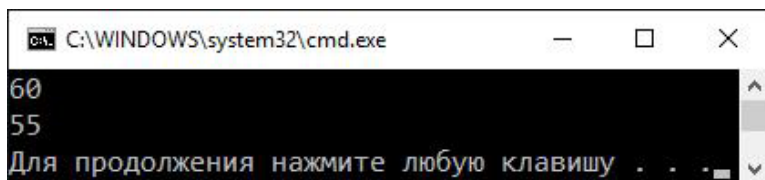


Рисунок 12.4. Вызов одноименных методов в различных классах

Эта ситуация не вызовет никаких претензий компилятора, поскольку, с точки зрения синтаксиса, он совершенно корректный. Однако возникает вопрос целесообразности создания первого пространства имён и всего его содержимого, поскольку `namespace A` никак не задействовано.

Ключевое слово `using`

Для того чтобы подключить в программу нужное пространство имён, необходимо использовать ключевое слово `using`, после которого указывается название пространства имён. В каждом Вашем консольном приложении присутствует строка:

```
using System;
```

Она нужна, чтобы Вы могли, к примеру, выводить на консоль текстовые сообщения:

```
public static void Main()
{
    Console.WriteLine(«Hello, everybody!»);
}
```

Дело в том, что класс `Console` размещён в пространстве имён `System`, и чтобы в программе не приходилось явно указывать название `System` при каждом обращении к этому классу, мы подключаем его через директиву `using`.

А вот как выглядит та же запись без применения директивы `using`:

```
public static void Main()
{
    System.Console.WriteLine("Hello, everybody!");
}
```

И так на протяжении всей программы. Утомительно, не так ли?

Конечно, при создании собственных пространств имён также может применяться ключевое слово `using`. Например, в нашем проекте находятся два файла — `Class1.cs` и `Class2.cs`, в каждом из них своё пространство имён, `CSharpNamespace1` и `CSharpNamespace2` соответственно. Мы хотим в методе класса из пространства имён `CSharpNamespace2` файла `Class2.cs` создать экземпляр класса из пространства имён `CSharpNamespace1` файла `Class1.cs`. Для этого нам всего-навсего нужно в файле `Class2.cs` подключить пространство имён из другого файла с помощью `using`. Вот как это выглядит:

```
// Class1.cs

using System;

namespace CSharpNamespace1
{
    class Foo
    {
        //...
    }
}
```

```
// Class2.cs

using System;
using CSharpNamespace1;
namespace CSharpNamespace2
{
```

```

class Class2
{
    public void Method()
    {
        Foo obj = new Foo();
    }
}

```

Если два пространства имён, указанные в операторах `using`, содержат тип с одинаковым именем, то в этом случае необходимо использовать полную (или как минимум более длинную) форму имени, чтобы компилятор знал, какой именно тип в каждом случае использовать. Если этого не сделать, компилятор не будет знать, к типу из какого пространства имён Вы обращаетесь в данный момент (Рисунок 12.5).

```

// Class1.cs

using System;
namespace NS
{
    public class Class
    {
        public int a = 1;
        public void Print()
        {
            Console.WriteLine("Printing from NS");
        }
    }
}

```

```
// Class2.cs

using System;
namespace NS1
{
    public class Class
    {
        public int b = 2;
        public void Print()
        {
            Console.WriteLine("Printing from NS1");
        }
    }
}
```

```
// Program.cs

using System;
using NS;
using NS1;
namespace M
{
    public class ClassM
    {
        public static void Main()
        {
            Class objA = new Class();
            Console.WriteLine("objA = " + objA.a);
            objA.Print();
            Class objB = new Class();
            Console.WriteLine("objB = " + objB.b);
            objB.Print();
        }
    }
}
```

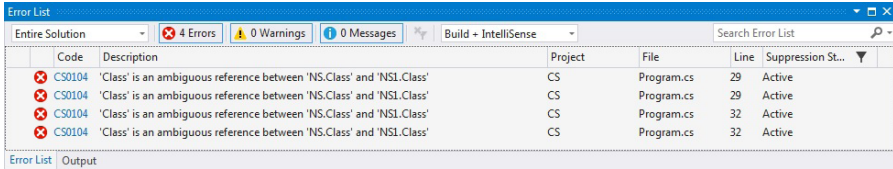



Рисунок 12.5. Ошибка: одноименные классы в различных пространствах имен

Чтобы исправить данные ошибки, нужно просто добавить названия пространств имён перед идентификаторами `Class`:

```
public static void Main()
{
    NS.Class objA = new NS.Class();
    Console.WriteLine("objA = " + objA.a);
    objA.Print();

    NS1.Class objB = new NS1.Class();
    Console.WriteLine("objB = " + objB.b);
    objB.Print();
}
```

Результат работы программы (Рисунок 12.6):

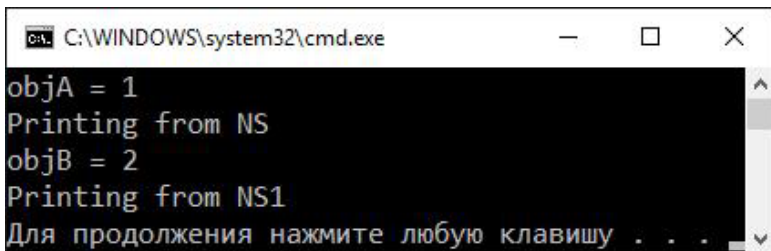


Рисунок 12.6. Явное указание пространств имен в классах

Объявление пространства имён

Для объявления пространства имён используется ключевое слово `namespace`, а всё, что определяется внутри него, заключается в фигурные скобки.

```
namespace dataBinding
{
    ...
}
```

В пространство имён могут включаться классы, структуры, делегаты, интерфейсы, перечисления, а также другие пространства имён (знакомство с некоторыми из вышеперечисленных средств языка C# Вам только предстоит в следующих уроках).

Вложенные пространства имен

Здесь всё предельно просто: пространства имён могут быть вложенными друг в друга. Это создаёт иерархические структуры для Ваших типов:

```
namespace ITAcademy
{
    namespace ProgrammingDepartment
    {
        namespace CSharp
        {
            namespace Basics
            {
                class MyClass
                {
                    // ...
                }
            }
        }
    }
}
```

```

    }
  }
}

```

Каждое имя в пространстве имён состоит из имён пространств, в которые оно вложено. Все имена разделяются между собой точками. Первым указывается самое внешнее имя, и далее вглубь по иерархии. Например, полным именем для класса из предыдущего примера будет:

```
ProgrammingDepartment.CSharp.Basics.MyClass
```

Иерархическое размещение характерно для .NET, и большая часть типов платформы имеет полное имя, состоящее из нескольких пространств имён.

```

System.Collections.Generic.List<>
System.Windows.Forms.Button
System.IO.StreamWriter
System.Xml.Serialization.XmlSerializationReader

```

Политика многоуровневой вложенности типов защищает от совпадения названий Ваших типов с типами, разработанными другими компаниями. Если Вы разрабатываете классы для своей компании, Microsoft рекомендует применять хотя бы двойной уровень вложенности пространств имён: первое — имя Вашей компании, а второе — название технологии или пакета программного обеспечения, к которому относится класс. Выглядит это примерно так:

```
namespace CompanyName
{
    namespace BankingSerices
    {
        class Client
        {
            //...
        }
    }
}
```

и полное имя типа будет:

```
CompanyName.BankingSerices.Client
```

В данном примере внешнее пространство имён `CompanyName` является корневым. Оно, как правило, служит лишь для расширения области видимости, и в нём непосредственно не определяются типы.

Полное имя пространства имен можно использовать в самом определении пространства имен, поэтому предыдущий пример можно переписать следующим образом:

```
namespace CompanyName.BankingSerices
{
    class Client
    {
        //...
    }
}
```

По поводу вложенности пространств имён нужно понимать одно простое правило: Вы создаёте области

видимости, вложенные друг в друга, а это значит, что все типы, объявленные в пространстве имён на ступеньку ниже, не доступны из объемлющих пространств имён. В подтверждение сказанного рассмотрим пример:

```
using System;
namespace NS
{
    namespace A
    {
        class Foo
        {
            public void Method()
            {
                Console.WriteLine("Hello from A.Foo");
            }

            static void Main()
            {
                Foo obj = new Foo();
                obj.Method();
            }
        }
    }
    class Foo
    {
        public void Method()
        {
            Console.WriteLine("Hello from NS.Foo");
        }
    }
}
```

Результатом этой программы будет запись, показанная на рисунке 12.7:

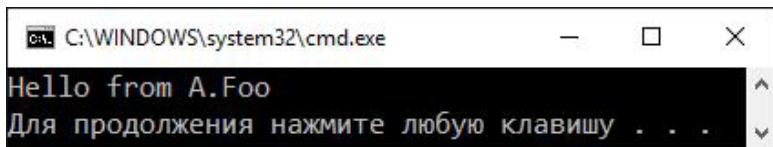


Рисунок 12.7. Область видимости пространств имен

Разбиение пространства имён на части

Представим ситуацию: несколько программистов Вашей фирмы работают над разработкой разных классов для одного приложения. Каждый из этих классов находится в своём файле, но помещён в одно и то же пространство имён. Когда эти файлы добавятся в проект, они будут скомпилированы в одну сборку.

Также Вы можете в пределах одного файла разделять пространство имён на части. Рассмотрим пример:

```
// Class1.cs
using System;
namespace A
{
    public class ClassA
    {
        public void Print()
        {
            Console.WriteLine("Printing from A.ClassA");
        }
    }
}
```

```
// Class2.cs
using System;
namespace A
{
```

```
class ClassB
{
    public void Print()
    {
        Console.WriteLine("Printing from A.ClassB");
    }
}
namespace A
{
    class ClassC
    {
        public void Print()
        {
            Console.WriteLine("Printing from A.ClassC");
        }
    }
}
```

```
// Program.cs

using System;
using A;
namespace B
{
    public class Class
    {
        public static void Main()
        {
            ClassA a = new ClassA();
            a.Print();

            ClassB b = new ClassB();
            b.Print();
        }
    }
}
```

```

        ClassC c = new ClassC();
        c.Print();
    }
}
}

```

Здесь пространство имён А распределено по двум файлам, а фактически разбито на три части, поскольку в файле Class2.cs оно дополнительно разделено на две части. В файле Program.cs включен оператор:

```
using A;
```

Это нужно для того, чтобы во втором пространстве имён В были доступны все типы, определённые в пространстве имён А. Результатом работы программы является следующее (Рисунок 12.8):

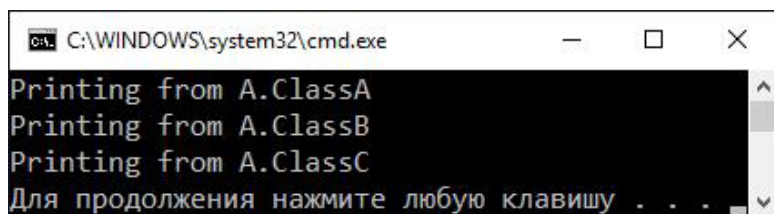


Рисунок 12.8. Пример разделения пространства имён на части

Пространство имён по умолчанию

По правилам платформы .NET, все имена должны быть объявлены в пределах пространства имён.

А что происходит, если мы не назначаем никакого пространства имён для своих типов?

В таком случае для них будет назначено пространство имён по умолчанию, и оно будет совпадать с именем проекта. Если Вас это не устраивает, тогда Вы можете изменить его имя на то, которое Вам нужно. Для этого на вкладке Application окна свойств проекта воспользуйтесь опцией Default namespace и укажите там имя, которое нужно (Рисунок 12.9).

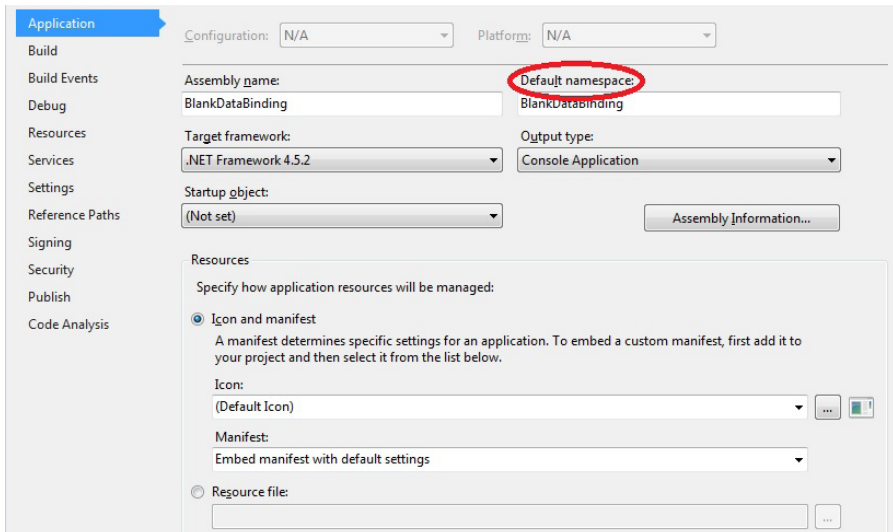


Рисунок 12.9. Настройка пространства имен по умолчанию

Если Вы так поступите, то каждый последующий файл, добавляемый вами в проект, автоматически будет помещаться в пространство имён с этим именем.

Директива псевдонима using

Ключевое слово `using` может применяться не только для подключения необходимого пространства имён, но также для создания псевдонимов пространств имён.

Чем это удобно? Поскольку, как мы упоминали ранее, пространство имён может иметь какую угодно глубину вложенности, соответственно размеры идентификатора могут быть очень громоздкими, например:

```
using System.Windows.Forms.DataVisualization.Charting;
```

Чтобы каждый раз не переписывать такое название, мы создаём, по сути, ещё одно, более лаконичное имя для пространства имён следующим способом:

```
using псевдоним = полное название пространства имён;
```

```
using WFCcharting = System.Windows.Forms.
    DataVisualization.Charting;
```

При этом в дальнейшем желательно обращаться к типу через квалификатор «::». В случае, если в данном пространстве имён будет определён тип с таким же именем, как псевдоним, конфликт возникать не будет. Рассмотрим следующий пример:

```
// Class1.cs
using System;
namespace NS
{
    public class Class
    {
        public void Print()
        {
            Console.WriteLine("Printing from NS.");
        }
    }
}
```

```
// Class2.cs
using System;
namespace NS1
{
    public class Class
    {
        public void Print()
        {
            Console.WriteLine("Printing from NS1.");
        }
    }
}
```

```
// Program.cs
using System;
using X = NS;
using Y = NS1;
namespace M
{
    public class ClassM
    {
        public int m = 3;
        public static void Main()
        {
            //X.Class objA = new X.Class(); Error
            X::Class objA = new X::Class();
            objA.Print();
            Y::Class objB = new Y::Class();
            objB.Print();
        }
    }
    public class X
    {
        //...
    }
}
```

Результат работы программы (Рисунок 12.10):

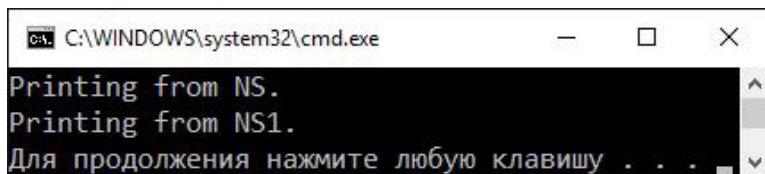


Рисунок 12.10. Использование квалификатора

В этом примере представлен проект, состоящий из нескольких файлов: `Class1.cs`, `Class2.cs`, `Program.cs`.

В каждом файле код заключён в своё пространство имён. В файле `Program.cs` назначаются псевдонимы для пространств имён из двух других файлов:

```
using X = NS;  
using Y = NS1;
```

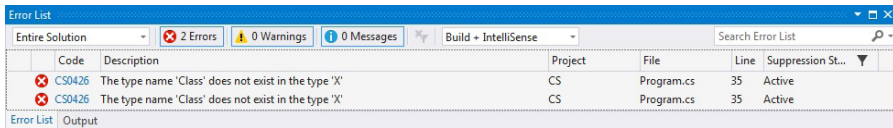
И здесь же через псевдонимы идёт обращение к типам из пространств имён `NS` и `NS1`. Казалось бы, ничем не примечательная ситуация, но...

В пространстве имён `M` файла `Program.cs` объявлен класс `X`, чьё имя совпадает с псевдонимом, назначенным для пространства имён `NS`:

```
public class X  
{  
    //...  
}
```

Если бы в теле метода `Main` мы обратились к идентификатору `X`, то компилятор расценил бы это как обращение к классу с одноимённым названием. Обратиться

через точку к классу Class из пространства имён X в этой ситуации невозможно, ошибка на рисунке 12.11.



Code	Description	Project	File	Line	Suppression St...
CS0426	The type name 'Class' does not exist in the type 'X'	CS	Program.cs	35	Active
CS0426	The type name 'Class' does not exist in the type 'X'	CS	Program.cs	35	Active

Рисунок 12.11. Ошибка на этапе компиляции

Ситуацию спасает квалификатор «::».

```
X::Class objA = new X::Class();
```

Использование using для подключения статических членов

При обращении к статическим членам класса необходимо указывать имя класса, например:

```
Console.WriteLine («Hello»);
```

И такие действия приходилось повторять на протяжении всей программы, но в версии C# 6.0 появилась возможность импортировать статические классы с помощью ключевого слова `using`, выглядит это следующим образом:

```
using static System.Console;
```

После этого вызов методов статических классов можно осуществить без указания имени класса, как показано ниже:

```

using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(«Корень квадратный из 81 = « +
Sqrt(81));
        WriteLine(«2 в степени 5 = « + Pow(2, 5));
    }
}

```

Результат работы кода:

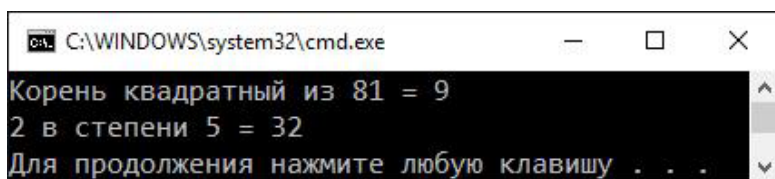


Рисунок 12.12. Использование директивы `using static`

Директива `using static` импортирует только доступные статические члены и вложенные типы, объявленные в указанном классе, унаследованные члены не импортируются.

Ключевое слово `using` также используется для создания оператора `using`, который помогает обеспечить правильную обработку объектов классов, реализующих интерфейс `IDisposable`, но знакомство с ним у Вас состоится в одном из следующих уроков.

13. Домашнее задание

1. Описать структуру `Article`, содержащую следующие поля: код товара; название товара; цену товара.
2. Описать структуру `Client` содержащую поля: код клиента; ФИО; адрес; телефон; количество заказов осуществленных клиентом; общая сумма заказов клиента.
3. Описать структуру `RequestItem` содержащую поля: товар; количество единиц товара.
4. Описать структуру `Request` содержащую поля: код заказа; клиент; дата заказа; перечень заказанных товаров; сумма заказа (реализовать вычисляемым свойством).
5. Описать перечисление `ArticleType` определяющее типы товаров, и добавить соответствующее поле в структуру `Article` из задания №1.
6. Описать перечисление `ClientType` определяющее важность клиента, и добавить соответствующее поле в структуру `Client` из задания №2.
7. Описать перечисление `PayType` определяющее форму оплаты клиентом заказа, и добавить соответствующее поле в структуру `Request` из задания №4.
8. Придумать класс, описывающий студента. Предусмотреть в нем следующие моменты: фамилия, имя, отчество, группа, возраст, массив (зубчатый) оценок по программированию, администрированию и дизайну. А также добавить методы по работе с перечисленными

данными: возможность установки/получения оценки, получение среднего балла по заданному предмету, распечатка данных о студенте.

9. Разработайте приложение «7 чудес света», где каждое чудо будет представлено отдельным классом. Создайте дополнительный класс, содержащий точку входа. Распределите приложение по файлам проекта и с помощью пространства имён обеспечьте возможность взаимодействия классов.
10. Разработать приложение, в котором бы сравнивалось население трёх столиц из разных стран. Причём страна бы обозначалась пространством имён, а город — классом в данном пространстве.

