Московский государственный технический университет
имени Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

Ю.Е. Гапанюк

СЕМИНАР №1
ПРОЕКТИРОВАНИЕ СИСТЕМЫ КЛАССОВ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Учебные материалы по дисциплине
«Архитектура автоматизированных систем обработки информации и управления»
Профиль: «Архитектура программных систем»

Москва – 2026

Оглавление

1 Краткая характеристика языка программирования C#

Современный объектно-ориентированный язык программирования общего назначения C# разработан компанией Microsoft для платформы .NET.

С его помощью можно разрабатывать консольные приложения, оконные приложения, веб-приложения, серверные API. Для обработки данных предназначены технологии LINQ и Entity Framework.

Первая версия C# появилась в начале 2000 годов и стех пор активно развивается. Создателем языка C# является Андерс Хейлсберг, который до работы над языком C# был разработчиком компилятора с языка Паскаль и среды разработки Delphi. Это безусловно сказалось на C#, который, не смотря на синтаксис, унаследованный от C++, впитал в себя лучшие структурные черты Паскаля.

2 Основные конструкции программирования языка C#

Данный раздел основан на **примере Structures**, приведенном целиком. В дальнейшем в основном будут даны только фрагменты примеров.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace Structures
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // явное объявление строки
            string str = "строка1";
```

```
            // компилятор выводит тип
            // по значению правой части выражения
            var str2 = "тоже строка";

            //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
            // Строковые интерполяции
            //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            string City = "Moscow";
            string Country = "Russia";

            //Старый способ вывода с помощью конкатенации строк
            Console.WriteLine(City + ", " + Country);
            //Старый способ вывода с помощью string.Format
            Console.WriteLine(string.Format("{0}, {1}", City, Country));
            //Новый способ вывода с помощью строковой интерполяции
            Console.WriteLine($"{City}, {Country}");


            //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
            // Условия
            //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            //Условный оператор (в отличие от C++ в условии
            //используется логический тип)
            if (str == "строка1")
            {
                Console.WriteLine("if: str == \"строка1\"");
            }
            else
            {
                Console.WriteLine("if: str != \"строка1\"");
            }

            //Условная операция
            string result = (str == "строка1" ? "Да" : "Нет");
            Console.WriteLine($"?: Равна ли строка {str} строке 'строка1'
- {result}");

            //Оператор switch
            string result2 = "";
            switch (str)
            {
                case "строка1":
                    result2 = "строка1";
                    break;

                case "строка2":
                case "строка3":
                    result2 = "строка2 или строка3";
                    break;

                default:
                    result2 = "другая строка";
                    break;
```

```
            }
            Console.WriteLine($"switch: {result2}");

            // Современный стиль (switch expression)
            // появился в C# 8
            string result21 = str switch
            {
                "строка1" => "строка1",
                "строка2" or "строка3" => "строка2 или строка3",  //
логический or
                _ => "другая строка"  // _ = default
            };

            // Более сложный пример с типами
            int? obj = 3;
            string description = obj switch
            {
                int i => $"Целое число: {i}",
                null => "Null",
            };

            // С условиями (guards)
            var number = 5;
            string category = number switch
            {
                < 0 => "Отрицательное",
                0 => "Ноль",
                > 0 and <= 10 => "Малое положительное",
                > 10 => "Большое положительное"
            };
            Console.WriteLine($"{result21} {obj} {number}");

            //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
            // Циклы
            //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            //Цикл for
            Console.Write("\nЦикл for: ");
            for (int i = 0; i < 3; i++)
                Console.Write(i);

            //Цикл foreach
            Console.Write("\nЦикл foreach: ");
            int[] array1 = { 1, 2, 3 };
            foreach (int i2 in array1)
                Console.Write(i2);

            //Цикл while
            Console.Write("\nЦикл while: ");
            int i3 = 0;
            while (i3 < 3)
            {
                Console.Write(i3);
                i3++;
            }
```

```
            //Цикл do while
            Console.Write("\nЦикл do while: ");
            int i4 = 0;
            do
            {
                Console.Write(i4);
                i4++;
            } while (i4 < 3);

            //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
            // Обработка исключений
            //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            Console.WriteLine("\n\nДеление на 0:");
            try
            {
                int num1 = 1;
                int num2 = 1;

                string zero = "0";
                int.TryParse(zero, out num2);

                int num3 = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Попытка деления на 0");
                Console.WriteLine(e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Собственное исключение");
                Console.WriteLine(e.Message);
            }
            finally
            {
                Console.WriteLine("Это сообщение выводится в блоке
finally");
            }


            Console.WriteLine("\nСобственное исключение:");
            try
            {
                throw new Exception("!!! Новое исключение !!!");
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Попытка деления на 0");
                Console.WriteLine(e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Собственное исключение");
```

```
            Console.WriteLine(e.Message);
        }
        finally
        {
            Console.WriteLine("Это сообщение выводится в блоке
finally");
        }

        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        // Константы
        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        const int int_const = 333;

        //Ошибка
        //int_const = 1;

        Console.WriteLine("Константа {0}", int_const);

        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        // Параметры функций
        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        //В C# по умолчанию аргументы обычных типов передаются по
значению, а объектов по ссылке
        //Аргументы ref всегда передаются по ссылке
        //Аргументы out являются только выходными параметрами

        string RefTest = "Значение до вызова функций";

        ParamByVal(RefTest);
        Console.WriteLine("\nВызов функции ParamByVal. Значение
переменной: " + RefTest);

        ParamByRef(ref RefTest);
        Console.WriteLine("Вызов функции ParamByRef. Значение
переменной: " + RefTest);

        int x = 2, x2, x3;
        ParamOut(x, out x2, out x3);
        Console.WriteLine("Вызов функции ParamOut. x={0}, x^2={1},
x^3={2}", x, x2, x3);

        //Объявление параметра x22 прямо в методе,
        //третий параметр не используется
        ParamOut(x, out int x22, out int _);

        //Переменное количество параметров
        ParamArray("Вывод параметров: ", 1, 2, 333);

        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        // Работа с null
        //+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        //Ошибка, типу int нельзя присвоить null
        //int nl = null;
```

```
            //Nullable value types
            //используют структуру Nullable<T>
            int? nl_number = 42;
            int? nl_nullNumber = null;

            if (nl_number.HasValue)
            {
                //присваивается значение int (42)
                int nl_value = nl_number.Value;
                Console.WriteLine(nl_value);
            }

            // Оператор ?? присваивает правое значение если левое null
            // Null-coalescing оператор ?? появился в C# 2
            int nl_result1 = nl_number ?? -1; // 42
            int nl_result2 = nl_nullNumber ?? -1; // -1
            Console.WriteLine($"{nl_number} {nl_nullNumber} {nl_result1}
{nl_result2}");

            //Nullable reference types – это ссылочные типы
            //(например, string), которые явно помечены как допускающие
null
            string? nl_str1 = "Строка1";
            string? nl_str2 = null;    // может быть null
            if(nl_str2 == null) Console.WriteLine(nl_str1);

            //Null-conditional оператор ?. появился в C# 6
            // Если nl_str2 == null, то возвращается null, ошибка не
возникает
            int ? nl_length1 = nl_str1?.Length;
            int? nl_length2 = nl_str2?.Length;
            Console.WriteLine($"{nl_length1} {nl_length2}");

            // Null-coalescing assignment (присваивание) ??=
            // появилось начиная с C# 8
            // Присваивание сработает, только если nl_str2 == null
            nl_str2 ??= "!!!";
            Console.WriteLine(nl_str2);

            //Для проверки на null можно использовать выражения
            // старый стиль
            if (obj == null) { }
            // современный стиль начиная с C# 7
            if (obj is null) { }
            if (obj is not null) { }
        }

        /// <summary>
        /// Передача параметра по значению
        /// </summary>
        /// <param name="param"></param>
        static void ParamByVal(string param)
        {
            param = "Это значение НЕ будет передано в вызывающую функцию";
```

```
        }

        /// <summary>
        /// Передача параметра по ссылке
        /// </summary>
        /// <param name="param"></param>
        static void ParamByRef(ref string param)
        {
            param = "Это значение будет передано в вызывающую функцию";
        }

        /// <summary>
        /// Выходные параметры объявляются с помощью out
        /// </summary>
        /// <param name="x"></param>
        /// <param name="x2"></param>
        /// <param name="x3"></param>
        static void ParamOut(int x, out int x2, out int x3)
        {
            x2 = x * x;
            x3 = x * x * x;
        }

        /// <summary>
        /// Переменное количество параметров задается с помощью params
        /// </summary>
        /// <param name="str"></param>
        /// <param name="ArrayParams"></param>
        static void ParamArray(string str, params int[] ArrayParams)
        {
            Console.Write(str);
            foreach (int i in ArrayParams)
                Console.Write(" {0} ", i);
        }
    }
}
```

На основе данного примера рассмотрим основные конструкции языка C#.

2.1 Пространства имен и сборки

Программа начинается с операторов using, каждый из которых указывает пространство имен для библиотечных классов. Любой класс в языке C# должен быть объявлен в каком-либо пространстве имен с использование оператора namespace.

Пространства имен представляют собой древовидную структуру. В каждой ее ветви содержатся вложенные классы и вложенные пространства

имен. Если с помощью оператора using подключаются классы какого-либо пространства имен, например

```
using System;
```

то классы вложенных пространств имен при этом автоматически не подключаются. Поэтому их необходимо подключать отдельными директивами:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Пространства имен представляют собой логическую структуру для систематизации классов. Выясним, как эта структура соотносится с откомпилированными классами.

Откомпилированный бинарный код для платформы .NET хранится в файлах сборок (assembly). Файл может иметь расширение .dll или .exe по аналогии с библиотеками и исполняемыми файлами ОС Windows. Но данные файлы содержат бинарный код, выполняющийся только на платформе .NET. Если же она не установлена, то ОС Windows не запустит такой исполняемый файл.

Файлы сборок следует подключать в разделе References проекта (рис. 3). В русской версии Visual Studio раздел References называется Ссылки. При нажатии правой кнопкой мыши на пункт References открывается диалог по добавлению новых сборок.

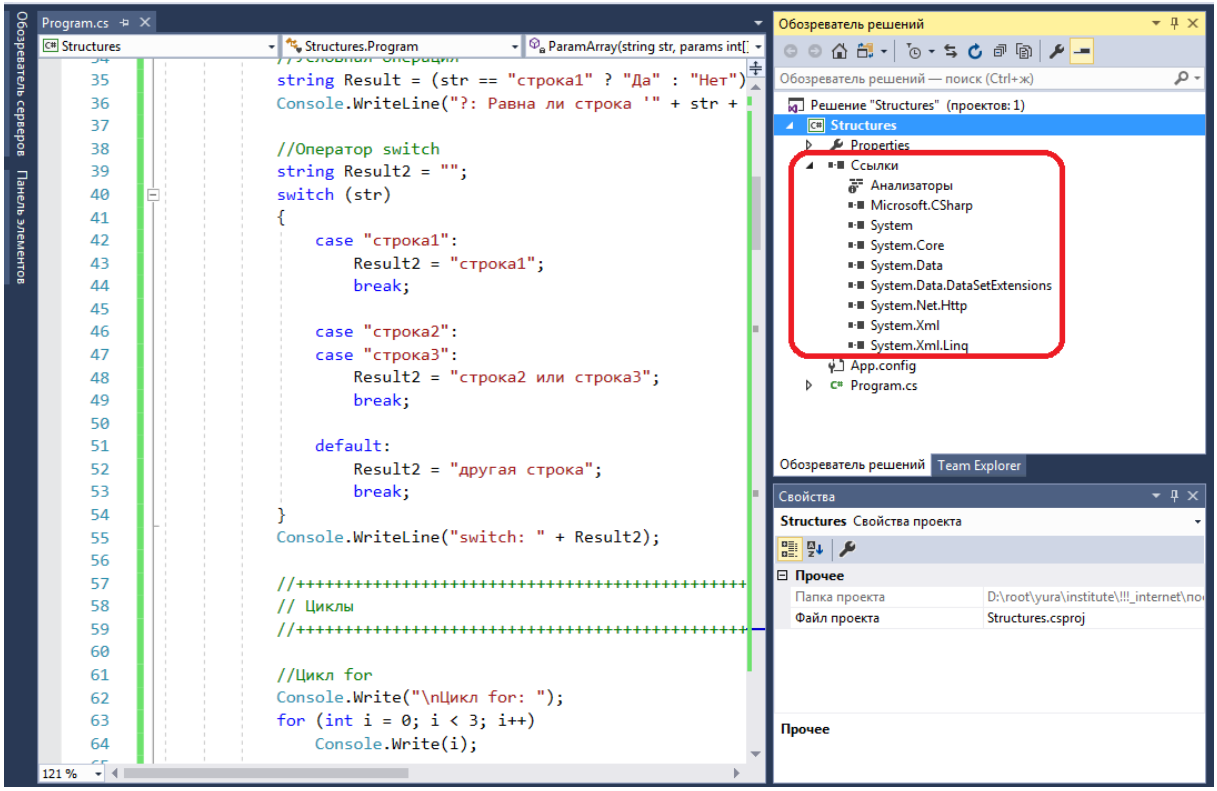

Рис. 1. Файлы сборок.

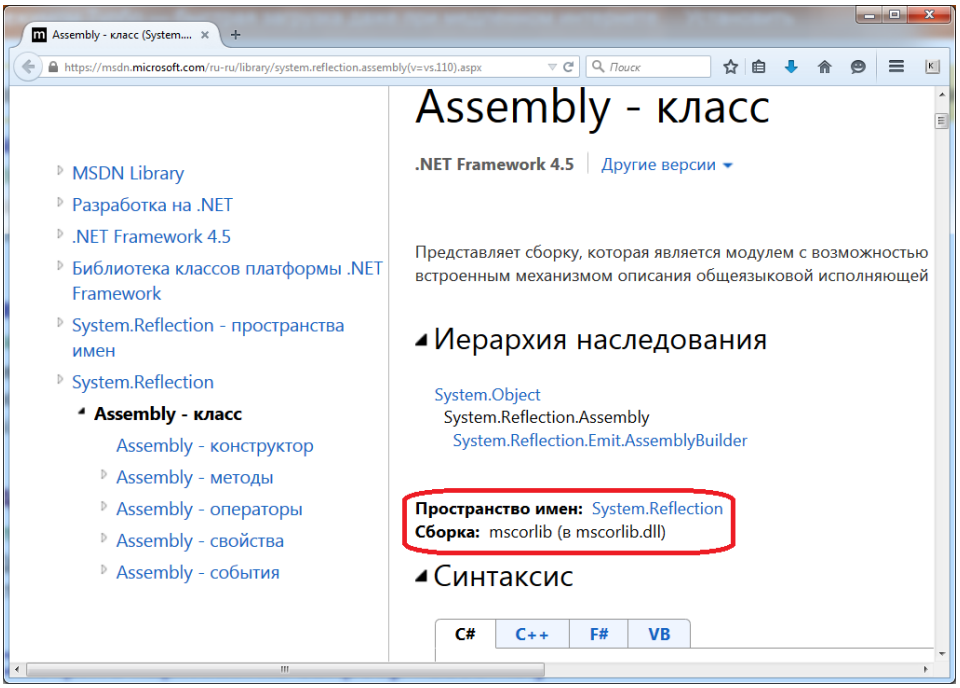

Рис. 2. Указание пространства имен и сборки в справочной системе.

Из рисунка 3 видно, что раздел References содержит те же значения что и операторы using. Однако это принципиально разная информация. В разделе References находятся имена физических файлов сборок – файлов,

которые содержат бинарный код, присоединяемый к проекту. Секция using ссылается на логическое название в дереве пространства имен.

При этом возможна ситуация, когда классы из одного и того же пространства имен находятся в разных сборках, и наоборот, одна сборка содержит классы из разных пространств имен. Поэтому в справочной системе Microsoft для каждого класса указано как имя сборки, так и пространство имен (рис 4).

Стоит отметить, что концепция пространства имен – специфика Microsoft.

В классическом языке C++ пространства имен отсутствуют, вместо них применяется подключение заголовочных файлов. Однако в версии языка C++, предлагаемой Microsoft, используется такой же механизм пространств имен как и в C#.

В Java существует концепция похожая на пространства имен – пакеты (package). По аналогии с пространством имен каждый класс может быть включен в пакет. Но в данной концепции существуют ограничения – пакет является каталогом файловой системы, в который вложены файлы классов. Если имя пакета составное (содержит точку), то это предполагает вложенность соответствующих каталогов. Один файл в Java не может содержать более одного класса, следовательно, файл-класс однозначно соответствует пакету-каталогу. Пространства имен в языке C# – более гибкий механизм, однако сторонники Java полагают, что подобная жесткость позволяет избежать ошибок и несоответствий, встречающихся в пространствах имен.

Далее в тексте программы приведены конструкции:

```
namespace Structures
{
    internal class Program
    {
    …
    }
}
```

Команда namespace объявляет пространство имен, а class – класс. Модификатор internal указывает что класс виден только в рамках текущей сборки. Команда namespace может содержать несколько классов. Чтобы сослаться на класс Program следует применить директиву:

```
using Structures;
```

Основной исполняемый метод консольного приложения – метод Main:

```
static void Main(string[] args)
  {
    …
  }
```

Строковый массив параметров метода args содержит параметры (аргументы) командной строки, которые могут быть заданы при вызове консольного приложения.

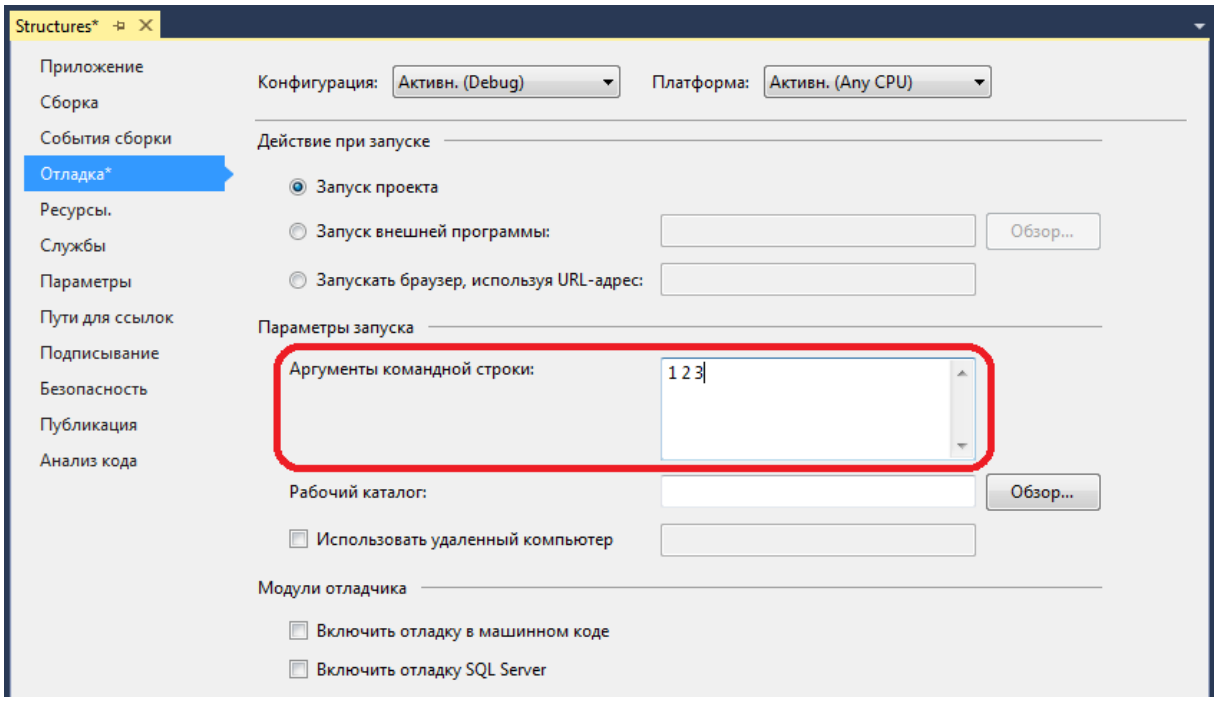

Рис. 3. Задание аргументов командной строки при отладке.

В Visual Studio в целях отладки данные параметры можно указать в свойствах проекта. Для этого нужно нажать правую кнопку мыши на названии проекта Structures, выбрать в контекстном меню пункт Свойства и установить параметры командной строки, что показано на рис. 5.

Далее в методе Main рассматриваются основные виды конструкций языка C#, которые очень похожи на соответствующие конструкции языков C++ и Java.

Замечание: Данный развернутый шаблон консольного приложения дан для пояснения смысла проекта. В современных проектах C# используется более компактный код, можно выделить два варианта:

1. Начиная с C# 10 (.NET 6) используется подход file-scoped namespaces (пространства имен с областью действия на файл).

Пример до C# 10:

```
using System;
namespace Structures
{
    internal class FileName
    {
        // код
    }
}
```

Пример начиная с C# 10:

```
using System;

namespace Structures;

internal class FileName
{
    // код
}
```

2. Начиная с C# 9.0 (.NET 5) можно писать код основной консольной программы без объявления класса и пространства имен. Такой шаблон используется по умолчанию в современных версиях Visual Studio.

Пример до C# 9:

```
using System;

namespace MyApp
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Пример начиная с C# 9:

```
using System;

Console.WriteLine("Hello, World!");
```

2.2 Строковая интерполяция

Строковая интерполяция (string interpolation) – это возможность упрощения синтаксиса, которая появилась в версии C# 6.

Строковая интерполяция позволяет указывать в строке шаблон для ее форматирования на основе переменных. Необходимо отметить, что похожие возможности есть в других языках программирования, в частности в Python и PHP.

Пример, использующий строковую интерполяцию:

```
string City = "Moscow";
string Country = "Russia";

//Старый способ вывода с помощью конкатенации строк
Console.WriteLine(City + ", " + Country);
//Старый способ вывода с помощью string.Format
Console.WriteLine(string.Format("{0}, {1}", City, Country));
//Новый способ вывода с помощью строковой интерполяции
Console.WriteLine($"{City}, {Country}");
```

Признаком использования строковой интерполяции является то, что при объявлении строки перед кавычками ставится символ доллара. Если в такой строке указано выражение в фигурных скобках, то оно автоматически вычисляется.

Необходимо отметить, что строковая интерполяция в настоящее время является основным способом соединения строк при консольном выводе.

Результаты вывода в консоль:

```
Moscow, Russia
```

```
Moscow, Russia
Moscow, Russia
```

2.3 Условные операторы и операторы сопоставления с образцом

Пример условного оператора if:

```
if (str == "строка1")
{
    Console.WriteLine("if: str == \"строка1\"");
}
else
{
    Console.WriteLine("if: str != \"строка1\"");
}
```

В качестве условия проверяется значение логического типа, который отсутствует в стандартном C++, но есть в Паскале, Java, C#.

Условный оператор вопрос-двоеточие выполняется аналогично тому, что в C++ и Java. До знака вопроса указывается логическое выражение. Если оно истинно, то выполняется код от вопроса до двоеточия, если ложно – код после двоеточия.

Пример оператора вопрос-двоеточие:

```
string Result = (str == "строка1" ? "Да" : "Нет");
```

Оператор switch выполняется аналогично тем, что в C++ и Java.

Пример оператора switch:

```
switch (str)
{
    case "строка1":
        Result2 = "строка1";
        break;

    case "строка2":
    case "строка3":
        Result2 = "строка2 или строка3";
        break;

    default:
        Result2 = "другая строка";
        break;
}
```

В круглых скобках после switch указывается проверяемое выражение, а после case – возможные варианты проверяемых значений.

Если значения после switch и case совпадают, то выполняются операторы, указанные в case до первого оператора break. Если оператор break не указан, то выполняются операторы следующей по порядку секции case, поэтому обычно каждую секцию case завершает оператор break. Если ни один оператор case не удовлетворяет условию, то выполняются операторы секции default.

Операторы сопоставления с образцом (pattern matching) – это классическая особенность языков программирования, использующих функциональный подход, таких как F#, Scala, Erlang, Haskell.

В языке C# данная возможность появилась, начиная с версии 7. Сопоставление с образцом похоже на условные операторы и поэтому построено на их основе.

Пусть дан массив объектов, который содержит строки и целые числа:

```
object[] array1 = { 1, "строка 1", 2, "строка 2", 3 };
```

Для того чтобы расширить условный оператор до оператора сопоставления с образцом в язык C# была добавлена конструкция is.

Пример сопоставления с образцом на основе условного оператора if:

```
foreach(object obj in array1)
{
    if(obj is int i1)
    {
        Console.WriteLine("Число -> " + i1.ToString());
    }
    else if (obj is string s1)
    {
        Console.WriteLine("Строка -> " + s1);
    }
}
```

В данном примере в цикле foreach по очереди перебираются элементы массива.

Если элемент массива соответствует целому числу «obj is int i1», то он автоматически приводится к целому типу и помещается в переменную i1.

Далее с ним можно выполнять необходимые действия, в примере это вывод в консоль.

Если элемент массива соответствует строке «obj is string s1», то он автоматически приводится к строке и помещается в переменную s1.

Результаты вывода в консоль:

```
Число -> 1
Строка -> строка 1
Число -> 2
Строка -> строка 2
Число -> 3
```

Вторым вариантом сопоставления с образцом в C# является использование оператора switch.

Пример сопоставления с образцом на основе оператора switch:

```
foreach (object obj in array1)
{
    switch(obj)
    {
        case string s1:
            Console.WriteLine("Строка -> " + s1);
            break;
        case int i1 when i1 > 2:
            Console.WriteLine("Число большее 2 -> " +
i1.ToString());
            break;
        case int i1:
            Console.WriteLine("Число -> " + i1.ToString());
            break;
    }
}
```

В этом случае в каждом операторе case указывается проверяемый тип и переменная, в которую сохраняется результат приведения типа.

С использованием ключевого слова when можно задавать так называемые «охранные выражения» (guards), которые накладывают дополнительные ограничения на проверку. В данном примере возможность приведения к целому типу проверяется дважды, случай i1>2 рассматривается отдельно и возникновение такого случая проверяется с помощью охранного выражения.

Результаты вывода в консоль:

```
Число -> 1
Строка -> строка 1
Число -> 2
Строка -> строка 2
Число большее 2 -> 3
```

Современный стиль сопоставления с образцом это switch expression (конструкция появилась в C# 8). Данная конструкция очень напоминает классические конструкции pattern matching из функциональных языков программирования.

```
// Современный стиль (switch expression)
// появился в C# 8
string result21 = str switch
{
    "строка1" => "строка1",
    "строка2" or "строка3" => "строка2 или строка3",  // логический
or
    _ => "другая строка"  // _ = default
};

// Более сложный пример с типами
int? obj = 3;
string description = obj switch
{
    int i => $"Целое число: {i}",
    null => "Null",
};

// С условиями (guards)
var number = 5;
string category = number switch
{
    < 0 => "Отрицательное",
    0 => "Ноль",
    > 0 and <= 10 => "Малое положительное",
    > 10 => "Большое положительное"
};
Console.WriteLine($"{result21} {obj} {number}");
```

Результаты вывода в консоль:

```
строка1 3 5
```

2.4 Операторы цикла

Счетный цикл for такой же как в C++ и Java. В круглых скобках после for указываются три оператора – начальное присвоение значения переменной цикла; условие выхода из цикла; оператор, который выполняется при переходе к следующему шагу цикла. Операторы разделяются точкой с запятой.

Пример цикла for:

```
for (int i = 0; i < 3; i++)
    Console.Write(i);
```

В данном примере оператор вывода переменной i не вложен в фигурные скобки, использование скобок не является обязательным, так как это единственный оператор цикла. Если бы в цикле выполнялось несколько действий, то их необходимо было бы вложить в фигурные скобки.

В языке C# также существует форма счетного цикла, предназначенная для перебора коллекции – foreach. В классическом языке C++ такой цикл отсутствует (однако он появился в новых версиях). В языке Java этот цикл имеет синтаксис for(переменная : коллекция).

Пример цикла foreach:

```
int[] array1 = { 1, 2, 3 };
foreach(int i2 in array1)
    Console.Write(i2);
```

В данном примере int i2 является объявлением переменной цикла, in отделяет переменную цикла от имени коллекции.

Следует отметить, что цикл foreach является одним из наиболее часто используемых видов цикла в C#. Это обусловлено тем, что в языке C# существует развитый набор коллекций, а реализация многих алгоритмов связана с перебором коллекций.

Циклы с предусловием и постусловием также очень похожи на те, что в C++ и Java.

Пример цикла с предусловием:

```
int i3 = 0;
while (i3 < 3)
{
    Console.Write(i3);
    i3++;
}
```

Пример цикла с постусловием:

```
int i4 = 0;
do
{
    Console.Write(i4);
    i4++;
} while (i4 < 3);
```

Операторы break и continue используются аналогично языкам C и C++.

2.5 Обработка исключений

Если потребовалось бы написать на чистом языке C приложение, выполняющее критически важные действия, то оно было бы написано в следующем стиле: выполняется вызов функции, функция возвращает код возврата (код ошибки), производится анализ кода возврата с помощью оператора switch…case, в зависимости от кода возврата выполняются дейтсвия по обработке ошибок. Не возникает сомнений в том, что такое программирование будет очень трудоемко.

Для решения данной проблемы в современных языках программирования (C#, Java, современные версии C++) существуют конструкции обработки исключений.

В блоке try записываются операторы, которые могут привести к возникновению ошибок. Собственно, термин «ошибка» употреблять не совсем корректно – то что в программе на чистом языке C было критической ошибкой теперь называют «исключением», исключительной ситуацией, которая может быть обработана в программе.

Если возникает ошибка (исключение) определенного вида, то она обрабатывается в блоке catch. Все исключения в C# – классы, унаследованные от класса Exception. В скобках после catch указывают класс исключения и переменную этого класса, из которой можно получить конкретные сведения о произошедшем исключении.

Блоков catch может быть несколько, каждый из них осуществляет перехват исключения определенного вида. В этом случае срабатывает только один, первый по порядку блок catch, поэтому последовательность обработки исключений очень важна. Необходимо, чтобы первыми располагались наиболее детальные исключения, те, которые размещаются на наиболее детальном уровне в дереве наследования от класса Exception. Если первым в списке поставить наиболее общий класс Exception, то любое исключение будет приведено к этому типу, и остальные блоки catch никогда не будут выполнены. Такое приведение типов является следствием полиморфизма классов в ООП – дочерний тип (класс) может быть приведен к родительскому типу (классу).

Необязательный блок finally располагается после блоков catch. Действия в блоке finally выполняются всегда, вне зависимости от того произошло исключение в блоке try или нет. Если, например, в блоке try происходит работа с файлами, то в блоке finally можно расположить оператор закрытия файла, который будет закрыт вне зависимости от того были ошибки при работе с файлом или нет.

Пример блока обработки исключений:

```
Console.WriteLine("\n\nДеление на 0:");
try
{
    int num1 = 1;
    int num2 = 1;

    string zero = "0";
    int.TryParse(zero, out num2);

    int num3 = num1 / num2;
}
```

```
catch (DivideByZeroException e)
{
    Console.WriteLine("Попытка деления на 0");
    Console.WriteLine(e.Message);
}
catch (Exception e)
{
    Console.WriteLine("Собственное исключение");
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("Это сообщение выводится в блоке finally");
}
```

В данном примере сначала обрабатывается детальное исключение деления на ноль (класс DivideByZeroException), а затем все остальные исключения, которые приводятся к наиболее общему классу Exception.

Кроме стандартных классов исключений, предусмотренных в .NET, разработчик может создавать собственные исключения, унаследованные от класса Exception. Однако такие исключения не будут вызываться автоматически, ведь никаких критичных действий вроде деления на ноль не происходит. Поэтому разработчик должен искусственно сгенерировать ошибку с помощью команды throw, причем можно генерировать и стандартные исключения, существующие в .NET.

Пример генерации исключения:

```
Console.WriteLine("\nСобственное исключение:");
try
{
    throw new Exception("!!! Новое исключение !!!");
}
catch (DivideByZeroException e)
{
    Console.WriteLine("Попытка деления на 0");
    Console.WriteLine(e.Message);
}
catch (Exception e)
{
    Console.WriteLine("Собственное исключение");
    Console.WriteLine(e.Message);
}
finally
{
```

```
    Console.WriteLine("Это сообщение выводится в блоке finally");
}
```

Если оператор throw используется без параметров в форме «throw;» то он сохраняет исходный stack trace что важно для детальной диагностики исключений.

В версии C# 6 появились также exception filters (оператор when), которые упростили проверку условий при обработке исключений.

Пример без exception filters:

```
catch (DivideByZeroException ex)
{
    if (attempt < 3)
        Console.WriteLine("Предупреждение 1");
    else
        Console.WriteLine("Предупреждение 2");
}
```

Пример с exception filters:

```
catch (DivideByZeroException) when (attempt < 3)
{
    Console.WriteLine("Предупреждение 1");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Предупреждение 2");
}
```

2.6 Оператор using и автоматическое освобождение ресурсов

Оператор using предназначен для работы с ресурсами, требующими освобождения. Например, это открытие файла, которое требует обязательного закрытия.

Оператор using используется в коде методов и его не следует путать с инструкцией using, которая используется в начале файла для подключения пространств имен.

```
// Фрагмент кода:
using (var file = File.OpenRead("data.txt"))
{
    Process(file);
}
```

```
// Компилятор превращает в:
var file = File.OpenRead("data.txt");
try
{
    Process(file);
}
finally
{
    if (file != null)
        ((IDisposable)file).Dispose();
}
```

Если файл не закрыть явным образом, то возникает ошибка.

Пример без использования using:

```
FileStream file = null;
try
{
    file = File.OpenRead("data.txt");
    // работа с файлом
}
finally
{
    file?.Dispose();  // освобождаем ресурс
}
```

Пример с использованием using:

```
using (FileStream file = File.OpenRead("data.txt"))
{
    // работа с файлом
}  // Dispose() вызывается автоматически здесь!
```

В C# 8 появилась конструкция using declaration. В этом случае using ставится перед присваиванием внутри метода.

```
void ModernUsing()
{
    using FileStream file = File.OpenRead("data.txt");

    // работа с файлом
    // ...

}  // Dispose() вызовется в конце области видимости
```

2.7 Вызов методов, передача параметров и возврат значений

Вызов методов происходит также как и в языках C++ и Java. Метод вызывается по имени, параметры передаются в круглых скобках после его имени. При объявлении метода указываются формальные параметры, при вызове в них выполняется подстановка фактических параметров, с которыми метод осуществляет работу при данном вызове.

В языке C# есть особенность, связанная с передачей параметров – ключевые слова ref и out.

Если перед параметром указывается ключевое слово ref, то значение передается по ссылке, то есть передается ссылка на параметр. Если параметр изменяется в методе, то эти изменения сохраняются в вызывающем методе.

При этом необходимо учитывать, что все значения ссылочных типов (объекты классов) автоматически передаются по ссылке даже без указания ключевого слова ref, их изменения в методе автоматически сохраняются.

Таким образом, ключевое слово ref актуально прежде всего для типов-значений. Оно является аналогом передачи параметра по указателю в языке C++, хотя в современных вариантах C++ также используется понятие ссылки.

Если перед параметром указывается ключевое слово out, то значение параметра обязательно должно быть инициализировано в вызываемом методе, что проверяется на этапе компиляции. До передачи в вызываемый метод значение переменной может быть не инициализировано. Инициализированное значение параметра становится доступно в вызывающем методе.

Ключевые слова ref и out должны быть указаны и при объявлении параметров в методе, и при вызове метода. Их указание при вызове метода, очевидно, излишне для компилятора, которому достаточно информации

при объявлении метода. Однако это позволяет программисту избежать ошибок, связанных с незапланированным изменением значений параметров в методе.

Пример вызова методов:

```
string RefTest = «Значение до вызова функций»;

ParamByVal(RefTest);
Console.WriteLine(«\nВызов функции ParamByVal. Значение переменной:
« + RefTest);

ParamByRef(ref RefTest);
Console.WriteLine(«Вызов функции ParamByRef. Значение переменной: «
+ RefTest);

int x = 2, x2, x3;
ParamOut(x, out x2, out x3);
Console.WriteLine(«Вызов функции ParamOut. X={0}, x^2={1}, x^3={2}»,
x, x2, x3);
```

Пример объявления методов:

```
/// <summary>
/// Передача параметра по значению
/// </summary>
static void ParamByVal(string param)
{
    param = «Это значение НЕ будет передано в вызывающую функцию»;
}

/// <summary>
/// Передача параметра по ссылке
/// </summary>
static void ParamByRef(ref string param)
{
    param = «Это значение будет передано в вызывающую функцию»;
}

/// <summary>
/// Выходные параметры объявляются с помощью out
/// </summary>
static void ParamOut(int x, out int x2, out int x3)
{
    x2 = x * x;
    x3 = x * x * x;
}
```

В версии языка C# 7 реализовано синтаксическое усовершенствование, которое позволяет объявлять out-параметры

непосредственно при вызове функции. Ранее для передачи out-параметров использовался следующий синтаксис:

```
int i;
OutFunction(out i);
```

Теперь можно использовать упрощенную конструкцию:

```
OutFunction(out int i);
```

Использование упрощенной конструкции позволяет существенно сократить код при большом количестве out-параметров.

Если какой-то out-параметр не используется, то его можно заменить на конструкцию discard (_):

```
OutFunction(out int i, out string j, out float j);
// результаты параметров j и k не нужны при вызове
OutFunction(out int i, out string _, out float _);
```

В метод может быть передано переменное количество параметров, для этого при объявлении параметра метода используется ключевое слово params.

Пример вызова метода с переменным количеством параметров:

```
ParamArray("Вывод параметров: ", 1, 2, 333);
```

Пример объявления метода с переменным количеством параметров:

```
/// <summary>
/// Переменное количество параметров задается с помощью params
/// </summary>
static void ParamArray(string str, params int[] ArrayParams)
{
    Console.Write(str);
    foreach (int i in ArrayParams)
        Console.Write(" {0} ", i);
}
```

Параметр с ключевым словом params должен быть объявлен последним в списке параметров.

Для возврата значений из методов, так же как и в языках C++ и Java, используется ключевое слово return.

2.8 Работа с null

```
//Ошибка, типу int нельзя присвоить null
int nl = null;
```

Nullable value types используют структуру Nullable<T>. Тип int? означает, что переменной можно присвоить или значение типа int или null.

```
int? nl_number = 42;
int? nl_nullNumber = null;

if (nl_number.HasValue)
{
    //присваивается значение int (42)
    int nl_value = nl_number.Value;
    Console.WriteLine(nl_value);
}
```

Null-coalescing оператор ?? появился в C# 2. Оператор ?? присваивает правое значение если левое null.

```
int nl_result1 = nl_number ?? -1; // 42
int nl_result2 = nl_nullNumber ?? -1; // -1
Console.WriteLine($"{nl_number} {nl_nullNumber} {nl_result1}
{nl_result2}");
```

Nullable reference types — это ссылочные типы (например, string), которые явно помечены как допускающие null.

```
string? nl_str1 = "Строка1";
string? nl_str2 = null;
if(nl_str2 == null) Console.WriteLine(nl_str1);
```

Null-conditional оператор ?. появился в C# 6. Если nl_str2 == null, то возвращается null, ошибка не возникает.

```
int ? nl_length1 = nl_str1?.Length;
int? nl_length2 = nl_str2?.Length;
Console.WriteLine($"{nl_length1} {nl_length2}");
```

Null-coalescing assignment (присваивание) ??= появилось начиная с C#8. Присваивание сработает, только если nl_str2 == null.

```
nl_str2 ??= "!!!";
Console.WriteLine(nl_str2);
```

Для проверки на null можно использовать выражения

```
// старый стиль
if (obj == null) { }
// современный стиль начиная с C# 7
if (obj is null) { }
if (obj is not null) { }
```

2.9 XML-комментарии

При объявлении классов, методов и других структур возможно задать комментарии к ним в виде XML-тэгов.

Такие комментарии сохраняются в откомпилированной сборке и используются Visual Studio для работы технологии дополнения кода IntelliSense.

Пример метода с XML-комментариями:

```
/// <summary>
/// Метод сложения двух целых чисел
/// </summary>
/// <param name="p1">Первое число</param>
/// <param name="p2">Второе число</param>
/// <returns>Результат сложения</returns>
static int Sum(int p1, int p2)
{
    return p1 + p2;
}
```

Перед XML-комментарием ставится три прямых слеша, далее указывается соответствующий тэг XML. Существует довольно большое количество XML-тэгов комментариев, но наиболее часто используются три из них:

- summary – краткое описание;
- param – описание входного параметра;
- returns – описание возвращаемого значения.

При работе в Visual Studio не нужно вводить полную структуру XML-комментариев. Чтобы добавить блок XML-комментариев, необходимо установить курсор на строку кода перед объявлением функции (класса и т.д.) и три раза набрать прямой слеш «/». После этого автоматически добавляется заголовок XML-комментария. Visual Studio автоматически определяет, для какой структуры добавляется XML-комментарий, и генерирует набор тэгов, подходящих для данного случая.

Пример автоматически сгенерированных XML-комментариев для метода:

```
/// <summary>
```

```
///
/// </summary>
/// <param name="p1"></param>
/// <param name="p2"></param>
/// <returns></returns>
static int Sum(int p1, int p2)
{
    return p1 + p2;
}
```

Информация о наборе параметров при генерации комментария формируется автоматически, однако в случае изменения параметров после добавления XML-комментария невозможна повторная генерация комментария, информация о новых параметрах должна быть добавлена в XML-комментарий вручную.

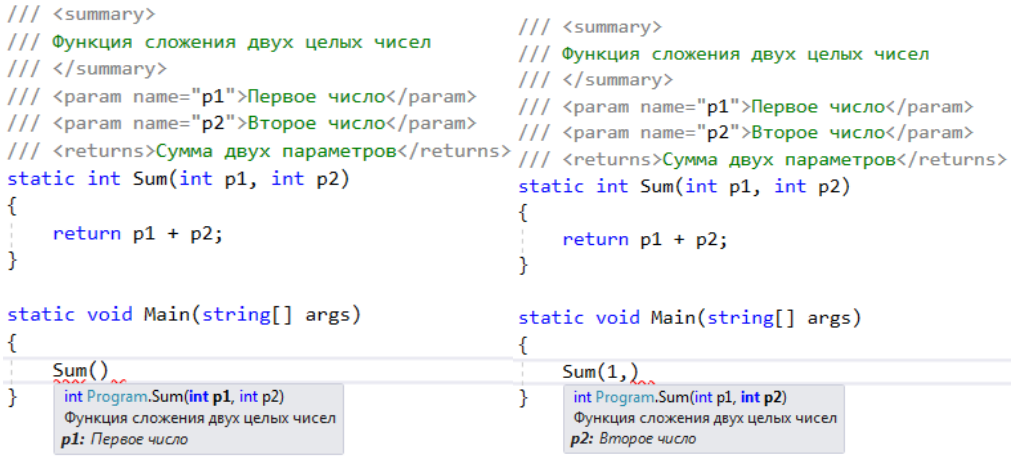

Рис. 4. Использование XML-комментариев при вызове функции Sum.

При вызове функции Sum на основе XML-комментариев будет автоматически сгенерирована подсказка, которая выводится с помощью IntelliSense (рис. 4).

Использование механизма XML-комментариев позволяет разработчику создавать хорошо документируемый код, при этом документация автоматически используется механизмом IntelliSense при вызове кода. Поэтому использование XML-комментариев чрезвычайно желательно при разработке проектов.

2.10 Директивы препроцессора

В языке C# как и в C++ существуют директивы препроцессора, однако они применяются существенно реже чем в C++.

В языке C# можно использовать директивы #define и #undef для определения и удаления символа, #if, #elif, #else, #endif – для организации условий, #warning и #error – для организации выдачи предупреждений и ошибок во время компиляции.

Примеры использования перечисленных директив:

```
#define debug

#if debug
#warning "Это предупреждение выдается в режиме debug"

#elif release
#error "Эта ошибка выдается в режиме release"

#else
#error "Должны быть включены debug или release"

#endif

#undef debug
```

Директивы #region и #endregion задают блок кода, который в Visual Studio может быть свернут или развернут.

Пример использования директив #region и #endregion:

```
#region Блок кода

    int int1 = 10;

#endregion
```

Результат работы директивы region в развернутом и свернутом виде показан на рис. 7.

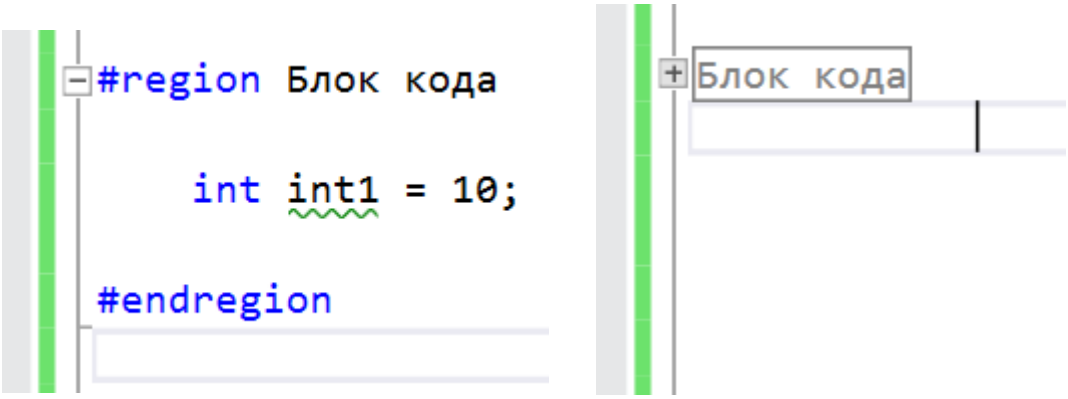

Рис. 5. Результат работы директивы region в развернутом и свернутом виде.

3 Основы объектно-ориентированного программирования в C#

Если языки C++ и Java довольно схожи в плане синтаксиса основных конструкций, то в плане объектно-ориентированного программирования (ООП) они довольно сильно различаются.

Подход к ООП в языке C# в целом намного ближе к Java чем к C++. Как и в Java в языке C# нет множественного наследования классов, оно реализуется с помощью интерфейсов.

Разберем основы ООП в языке C# на основе фрагментов **примера Classes**, которые рассматриваются далее в этом разделе.

3.1 Объявление класса и его элементов

Классы в языке C# объявляются с использованием ключевого слова class.

Рассмотрим более детально базовый класс примера – BaseClass:

```
/// <summary>
/// Базовый класс
/// </summary>
class BaseClass
{
    //Вместо полей данных рекомендуется использовать свойства
    private int i;
```

```
    //Конструктор
    public BaseClass(int param) { this.i = param; }
    //Методы с различными сигнатурами
    public int MethodReturn(int a) { return i; }
    public string MethodReturn(string a) { return i.ToString(); }

    //Свойство
    //private-значение, которое хранит данные для свойства
    private int _property1 = 0;
    //объявление свойства
    public int property1
    {
        //возвращаемое значение
        get { return _property1; }
        //установка значения, value - ключевое слово
        set { _property1 = value; }
        //private set { _property1 = value; }
    }

    /// <summary>
    /// Вычисляемое свойство
    /// </summary>
    public int property1mul2
    {
        get { return property1 * 2; }
    }

    //Автоматически реализуемые свойства
    //поддерживающая переменная создается автоматически
    public string property2 { get; set; }
    public float property3 { get; private set; }
}
```

3.2 Объявление конструктора

Класс содержит конструктор:

```
public BaseClass(int param) { this.i = param; }
```

Имя конструктора совпадает с именем класса. Конструктор принимает один параметр и присваивает его переменной класса, доступ к которой выполняется с помощью ключевого слова this. Если конструктор не определен в классе явно, то считается, что у него есть пустой конструктор без параметров.

3.2.1 Объявление методов

Также класс содержит методы с одинаковыми именами, но различными сигнатурами, что допустимо в языке C#:

```
public int MethodReturn(int a) { return i; }
public string MethodReturn(string a) { return i.ToString(); }
```

Как и в Java, в языке C# модификаторы видимости указываются перед каждым элементом класса, в языке C++ они задаются в виде секций с двоеточием, например «public:».

В языке C# используются следующие модификаторы видимости:

- public – элемент виден и в классе и снаружи класса;
- private – элемент виден только в классе;
- protected – элемент виден только в классе и наследуемых классах;
- internal – элемент виден в текущей сборке;
- protected internal – элемент виден в классах текущей сборки или в наследуемых классах, даже если наследуемые классы находятся в другой сборке.
- private protected – элемент виден только в наследуемых классах текущей сборки.
- file – применяется только к типам верхнего уровня (классам, структурам, интерфейсам), а не к членам класса. Делает этот тип видимым только в текущем файле.

3.3 Объявление свойств

Важные понятие языка C# – свойства. Оно отсутствует в языках C++ и Java в явном виде, вернее реализуется в этих языках с помощью данных и методов.

Если бы в языке C# не было свойств, также как и в C++ и Java, то можно было бы написать следующий код:

```
//объявление переменной
private int i;
//метод чтения
public int get_i() { return this.i; }
//метод записи
public void set_i(int value) { this.i = value; }
```

Смысл этого кода вполне понятен – к закрытой переменной можно обратиться на чтение и запись с помощью открытых методов. Но в языке C# для реализации такой задачи существует специальный вид конструкции – свойство.

Пример объявления простого свойства:

```
//private-значение, которое хранит данные для свойства
private int _property1 = 0;

//объявление свойства
public int property1
{
    //возвращаемое значение
    get { return _property1; }
    //установка значения, value - ключевое слово
    set { _property1 = value; }
}
```

Переменная _property1 является закрытой (private) и содержит данные для свойства. Такую переменную принято называть опорной переменной свойства. Как правило, опорные переменные всегда имеют область видимости private или protected, чтобы к ним не было внешнего доступа. Для опорных переменных принято следующее соглашение по наименованию – опорная переменная имеет то же имя что и свойство, но перед ним ставится подчеркивание «_».

Далее следует объявление свойства «public int property1». Внешне оно очень похоже на объявление переменной, но после него ставятся фигурные скобки и указываются секции get и set, которые принято называть аксессорами (accessors), поскольку они обеспечивают доступ к свойству.

Get-аксессор (аксессор чтения) – метод, который вызывается при чтении значения свойства. Как правило, он содержит конструкцию return,

возвращающую значение опорной переменной. Конечно, в get-аксессоре может выполняться и более сложный код.

Тип возвращаемого значения get-аксессора, который не задается в коде в явном виде, совпадает с типом свойства.

Set-аксессор (аксессор записи) является методом, который вызывается при присвоении значения свойству. Как правило, он содержит оператор присваивания значения опорной переменной «_property1 = value». В этом случае ключевое слово value обозначает то выражение, которое стоит в правой части от оператора присваивания.

Тип возвращаемого значения set-аксессора, который не задается в коде в явном виде, это тип void.

Объявленное свойство ведет себя как переменная, которой можно присвоить или прочитать значение.

Пример использования свойства:

```
//Создание объекта класса для работы со свойством
BaseClass bc = new BaseClass(333);
//Присвоение значения свойству (обращение к set-аксессору)
bc.property1 = 334;
//Чтение значения свойства (обращение к get-аксессору)
int temp = bc.property1;
```

Таким образом, свойство «кажется» обычной переменной, которой можно присвоить или прочитать значение. Однако при этом происходит обращение к соответствующим аксессорам свойства.

Области видимости аксессоров по умолчанию совпадают с областью видимости свойства, как правило, используется область видимости public. Однако для каждого аксессора можно указать свою область видимости, например:

```
public int property1
{
    get { return _property1; }
    private set { _property1 = value; }
}
```

В этом случае область видимости аксессора чтения совпадает с областью видимости свойства (public), а область видимости аксессора записи ограничена текущим классом (private). То есть прочитать значение такого свойства можно из любого места программы, а присвоить – только в текущем классе.

Рассмотренный код аксессоров используется очень часто. Чтобы облегчить написание кода, в языке C# для таких «стандартных» свойств принята упрощенная форма синтаксиса, называемая автоопределяемым свойством:

```
public string property2 { get; set; }
```

Такая упрощенная форма синтаксиса эквивалентна следующему описанию:

```
private string _property2;

public string property2
{
    get { return _property2; }
    set { _property2= value; }
}
```

В автоопределяемом свойстве опорная переменная «_property2» в исходном коде явно не задается, а автоматически генерируется на этапе компиляции и позволяет хранить значение свойства на этапе выполнения программы.

Для аксессоров как автоопределяемого свойства, так и обычного свойства, можно явно указывать собственные области видимости:

```
public float property3 { get; private set; }
```

Свойство может содержать только один из аксессоров. Например, можно создавать вычисляемые свойства, которые используются для вычисления значений и базируются на опорных переменных других свойств:

```
/// <summary>
/// Вычисляемое свойство
/// </summary>
public int property1mul2
```

```
{
    get { return property1 * 2; }
}
```

Вычисляемые свойства, как правило, содержат только аксессоры чтения. Следует отметить, что для данного свойства также задан XML-комментарий.

Для программистов, работающих на C++ или Java, может быть не совсем понятно, почему в языке C# свойствам придается такое важное значение. В языке C# в классах вообще не принято использовать public переменные, вместо них употребляются автоопределяемые свойства.

Свойства позволяют придавать программам на языке C# дополнительную гибкость, особенно при модификации кода. Например, если изменились правила вычисления какой-либо переменной (допустим, нужно возвращать значение переменной. умноженной на 2), то можно модифицировать get-аксессор не внося изменений в вызывающий код. Если же при присвоении переменной нужно производить дополнительный контроль (допустим, целочисленной переменной можно присваивать значения только от 1 до 1000), его можно поместить в set-аксессор (если условие не выполняется, то генерируется исключение) не внося изменений в вызывающий код.

Поэтому в языке C# принято объявлять public-переменные класса как автоопределяемые свойства, а при необходимости свойство может быть переписано в полной форме с расширением кода аксессоров.

В версии C# 6 появились новые возможности работы со свойствами.

При объявлении свойства возможна его непосредственная инициализация:

```
public string String1 { get; set; } = "строка 1";
```

Возможно объявление автоопределяемых свойств, предназначенных только для чтения. Такие свойства могут быть инициализированы в конструкторе класса или непосредственно при объявлении свойства.

```
public int Int1 { get; } = 333;
public int Int2 { get; }

/// <summary>
/// Конструктор класса
/// </summary>
public PropertyExample()
{
    this.Int2 = 45;
}
```

Попытка изменить такое свойство, например, в методе класса приводит к ошибке:

```
public void Method1()
{
    //Ошибка
    this.Int1 = 123;
}
```

Компилятор выдает следующий текст ошибки: «*Невозможно присвоить значение свойству или индексатору "PropertyExample.Int1" — доступ только для чтения*».

Expression-bodied свойства:

```
// Вместо:
public int property1mul3
{
    get { return property1 * 2; }
}

// Можно написать:
public int property1mul3 => property1 * 2;
```

В версии C# 7 появились expression-bodied аксессоры:

```
private int _value;

public int Value
{
    get => _value;
    // Валидация
    set => _value = value < 0 ? 0 : value;
}
```

В версии C# 9 появились Init-аксессоры, которые позволяют устанавливать значение только при инициализации объекта:

```
public class Person
{
```

```
    public string FirstName { get; init; }
    public string LastName { get; init; }
}

// Использование:
var person = new Person
{
    FirstName = "Иван",
    LastName = "Иванов"
};

// Ошибка компиляции:
// person.FirstName = "Петр";
```

C# 11 появились Required-свойства, которые обязывают инициализировать свойство при создании объекта:

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

//=========================================
//Required свойства
//=========================================

// Ошибка компиляции - не указано обязательное свойство Title:
// var book = new Book { Year = 2024 };

// Правильно:

var book1 = new Book
{
    Title = "Книга1",
    Year = 2026
};

var book2 = new Book
{
    Title = "Книга2",
};
```

3.4 Record-типы с неизменяемыми свойствами

В версии C# 9 появились Record-типы с неизменяемыми свойствами. Чаще всего используются для описания данных при взаимодействии с базами данных.

```
public record Point(int X, int Y);

// Пример 1: Создание и использование
var point1 = new Point(10, 20);
Console.WriteLine(point1); // Output: Point { X = 10, Y = 20 }
Console.WriteLine($"X = {point1.X}, Y = {point1.Y}"); // X = 10, Y =
20

// Пример 2: Value-based equality (сравнение по значению, а не по
ссылке)
var point2 = new Point(10, 20);
var point3 = new Point(15, 25);

Console.WriteLine(point1 == point2); // True (одинаковые значения)
Console.WriteLine(point1 == point3); // False (разные значения)

// Пример 3: Иммутабельность - попытка изменить вызовет ошибку
компиляции
// point1.X = 30; // ОШИБКА: init-only property

// Пример 4: Использование with-expression для создания копии с
изменениями
var point4 = point1 with { X = 100 };
Console.WriteLine(point1); // Point { X = 10, Y = 20 } (оригинал не
изменился)
Console.WriteLine(point4); // Point { X = 100, Y = 20 } (новый
объект)

// Пример 5: Деконструкция
var(x, y) = point1;
Console.WriteLine($"x = {x}, y = {y}"); // x = 10, y = 20

// Пример 6: Паттерн-матчинг
string Describe(Point p) => p switch
{
    { X: 0, Y: 0 } => "Начало координат",
    { X: 0 } => "На оси Y",
    { Y: 0 } => "На оси X",
    { X: var x, Y: var y } when x == y => "На диагонали",
    _ => "Обычная точка"
};
```

```
Console.WriteLine(Describe(new Point(0, 0))); // "Начало координат"
Console.WriteLine(Describe(new Point(5, 5)));  // "На диагонали"
```

3.5 Объявление статических элементов класса

Элемент класса в языке C# может быть объявлен как статический с помощью ключевого слова static. Это означает, что данный элемент (поле данных, свойство, метод) принадлежит не объекту класса, а классу в целом, и для работы с такими элементами не нужно создавать объект класса.

Вызов статического метода выполняется в формате «Имя_класса.имя_метода(параметры)». Таким образом, для вызова как статических, так и нестатических методов используется символ «.». Но в случае статического метода перед символом «.» ставится имя класса, а в случае нестатического метода перед символом «.» ставится имя объекта. Аналогично создаются и вызываются статические методы в Java, а в языке C++ вместо точки используется символ удвоенного двоеточия «::».

В языке C# весь класс может быть объявлен как статический. В этом случае ключевое слово static указывается перед именем класса, а все элементы класса должны быть статическими.

3.6 Наследование класса от класса

В языке C# наследовать класс можно только от одного класса. В данном примере класс ExtendedClass1 наследуется от класса BaseClass:

```
/// <summary>
/// Наследуемый класс 1
/// </summary>
class ExtendedClass1 : BaseClass
{
    private int i2;
    private int i3;

    //Конструкторы
    //base(pi) - вызов конструктора базового класса
```

```
    public ExtendedClass1(int pi, int pi2) : base(pi) { i2 = pi2; }

    //this(pi, pi2) - вызов другого конструктора этого класса
    public ExtendedClass1(int pi, int pi2, int pi3) : this(pi, pi2)
    { i3 = pi3; }

/// <summary>
/// Метод виртуальный, так как он объявлен в самом базовом классе
/// object
/// поэтому чтобы его переопределить добавлено ключевое слово
/// override
/// </summary>
    public override string ToString()
    {
        return "i=" + MethodReturn("1")
                + " i2=" + i2.ToString() + " i3=" + i3.ToString();
    }
}
```

Синтаксис наследования в языке C# аналогичен синтаксису языка C++. Для обозначения наследования используется символ двоеточия при объявлении класса, а затем указывается имя базового класса «class ExtendedClass1 : BaseClass».

3.7 Вызов конструкторов из конструкторов

При наследовании может возникнуть проблема, связанная с доступом к private полям базового класса, и в первую очередь она может появиться в конструкторе. Наследуемый класс имеет доступ к protected полям базового класса, но не имеет доступа к private полям. Инициализация private полей может быть реализована с помощью вызова конструктора базового класса:

```
public ExtendedClass1(int pi, int pi2) : base(pi)
{
    i2 = pi2;
}
```

Ключевое слово base обозначает вызов конструктора базового класса. В качестве параметров в круглых скобках указываются параметры, передаваемые в конструктор базового класса. После вызова конструктора базового класса (инициализация private полей базового класса)

выполняются действия указанные в фигурных скобках в теле текущего конструктора.

Аналогично может быть вызван другой конструктор текущего класса, но в этом случае вместо ключевого слова base используется ключевое слово this.

Пример использования ключевого слова this:

```
public ExtendedClass1(int pi, int pi2, int pi3) : this(pi, pi2)
{
    i3 = pi3;
}
```

В этом случае посредством ключевого слова this вызывается рассмотренный ранее конструктор класса с двумя параметрами «public ExtendedClass1(int pi, int pi2)», затем выполняются действия указанные в фигурных скобках в теле текущего конструктора.

3.8 Виртуальные методы

Пример переопределения виртуального метода ToString:

```
public override string ToString()
{
    return "i=" + MethodReturn("1")
             + " i2=" + i2.ToString() + " i3=" + i3.ToString();
}
```

Метод ToString объявлен в классе Object, который является базовым в иерархии классов C#.

Подход к виртуальным методам в языках Java и C# существенно различается.

В Java все нестатические метода класса – виртуальные. Разработчики языка C# полагают, что это приводит к снижению производительности, так как адрес для выполнения виртуального метода должен динамически вычисляться на этапе выполнения.

Поэтому в языке C# виртуальные методы должны быть явно объявлены с помощью ключевых слов virtual или abstract (абстрактные

методы – аналог чистых виртуальных методов в C++). Детали объявления абстрактных методов будут рассмотрены ниже.

Для переопределения виртуального метода необходимо использовать ключевое слово override. Очевидно, что его применение избыточно, ведь компилятор и так «знает» что в базовом классе объявлен виртуальный метод с таким именем.

Здесь, как и в случае передачи параметров ref и out используется синтаксис языка C#, «дисциплинирующий» разработчика. Разработчик должен явно указать, что он «понимает» что реализует данный метод как виртуальный.

Если пропустить объявление ключевого слова «override», то компилятор выдает не ошибку, а предупреждение, приведенное на рис 8.

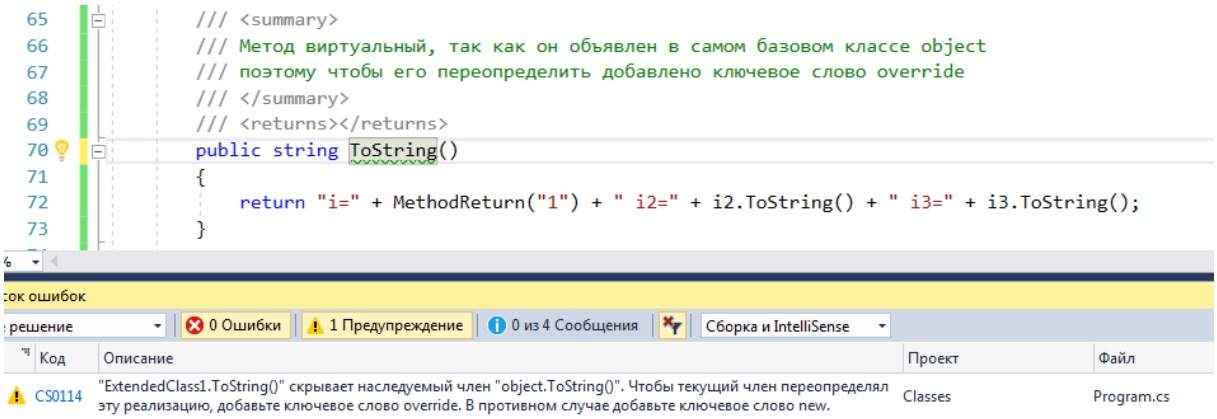

Рис. 6. Предупреждение в случае отсутствия ключевого слова «override».

Текст предупреждения: *«"Classes.ExtendedClass1.ToString()" скрывает наследуемый член "object.ToString()". Чтобы текущий член переопределял эту реализацию, добавьте ключевое слово override. В противном случае добавьте ключевое слово new.»*.

Таким образом, компилятор требует, чтобы программист явно указал ключевое слово override, если он считает, что данный метод следует использовать как виртуальный. Если программист хочет показать что данный метод не должен применяться как виртуальный, но имеет имя,

совпадающее с именем виртуального метода, то он должен указать ключевое слово new.

Принято считать, что при использовании ключевого слова new происходит «сокрытие» виртуального метода. В этом случае код будет выглядеть так:

```
public new string ToString()
{
    return "i=" + MethodReturn("1")
             + " i2=" + i2.ToString() + " i3=" + i3.ToString();
}
```

В теле метода формируется текстовая строка, которая возвращается с помощью оператора return. Для конкатенации фрагментов строки используется оператор «+». Метод MethodReturn, возвращающий строковое значение, унаследован из базового класса. Выражение «i2.ToString()» возвращает строковое представление целочисленной переменной i2.

3.9 Абстрактные классы и методы

В языке C# как и в языках C++ и Java можно объявлять абстрактные классы.

Следует напомнить, что абстрактным считается класс, который содержит хотя бы один виртуальный метод без реализации, такой метод должен быть переопределен в наследуемом классе. Абстрактные классы используются в цепочке наследования, но создавать объекты абстрактных классов нельзя.

В языке C++ абстрактным считается класс, содержащий хотя бы одну чистую виртуальную функцию. Это такая функция, которой присваивается значение 0, пример: «virtual void Function1() = 0;». Никаких специальных объявлений на уровне класса при этом не делается.

В языках Java и в C# чистые виртуальные методы называются абстрактными, перед их объявлением указывается ключевое слово abstract.

Если хотя бы один из методов объявлен абстрактным, то весь класс должен быть объявлен как абстрактный, поэтому ключевое слово abstract также должно быть указано при объявлении класса.

Пример класса геометрической фигуры:

```
/// <summary>
/// Класс геометрической фигуры
/// </summary>
abstract class Figure
{
    /// <summary>
    /// Тип фигуры
    /// </summary>
    public string Type { get; set; }

    /// <summary>
    /// Вычисление площади
    /// </summary>
    public abstract double Area();

    /// <summary>
    /// Приведение к строке, переопределение метода Object
    /// </summary>
    public override string ToString()
    {
        return this.Type + " площадью " + this.Area().ToString();
    }
}
```

Класс содержит абстрактный метод вычисления площади:

```
public abstract double Area();
```

Объявление метода как абстрактного автоматически делает его виртуальным, поэтому в наследуемом классе абстрактный метод должен переопределяться с ключевым словом override.

Поскольку класс Figure содержит хотя бы один абстрактный метод, то весь класс также объявлен как абстрактный.

В случае наследования от абстрактных классов Visual Studio позволяет автоматически генерировать заглушки для абстрактных методов. Создадим класс прямоугольник, наследуемый от фигуры. В этом случае при нажатии правой кнопки мыши на имени базового класса в

контекстном меню появляется пункт автоматической реализации абстрактного класса (рис 9).

Если выбрать данный пункт меню, то будет сгенерирован следующий код:

```
class Rectangle2 : Figure
    {
        public override double Area()
        {
            throw new NotImplementedException();
        }
    }
```

В наследуемый класс добавлена «заглушка» метода. Метод правильно объявлен, но вместо реализации генерируется исключение NotImplementedException.

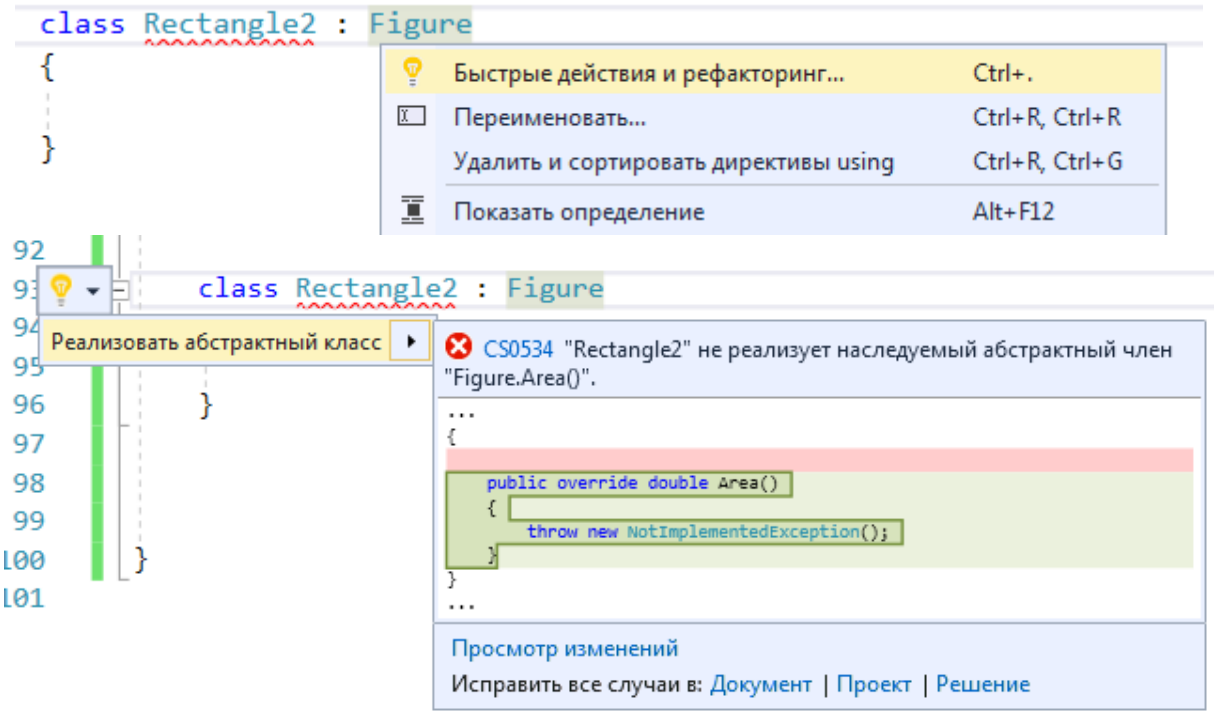

Рис. 7. Реализация абстрактного класса.

3.10 Интерфейсы

Абстрактный класс может содержать как абстрактные, так и обычные методы. Обычные методы содержат реализацию в виде кода языка C# и могут быть вызваны в наследуемом классе.

Интерфейс является «предельным случаем» абстрактного класса и не содержит реализации.

Интерфейс может содержать только объявления методов и свойств без указания области видимости.

В языке C# допустимо наследование от нескольких интерфейсов, но только от одного класса.

Наследование от нескольких классов может порождать ряд проблем, из которых наиболее известной является проблема «ромбовидного наследования». На рис 10 приведена диаграмма наследования, представляющая собой ромб. В этом случае классы B и C наследуются от класса A. Класс D наследуется от классов B и C.

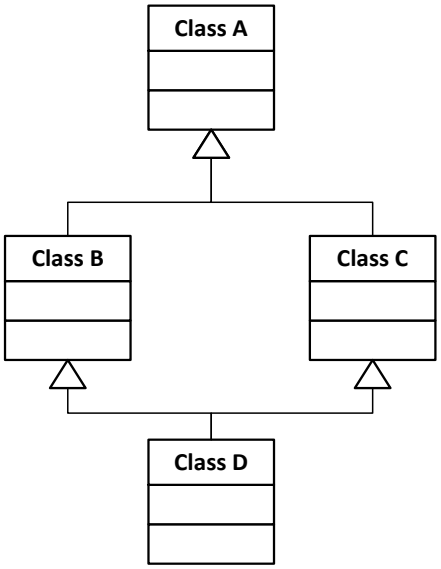

Рис. 8. Пример ромбовидного наследования.

При этом возникает вопрос о том, что делать с полями класса A. Они дублируются при наследовании в классах B и C и могут изменяться различным образом. Класс D может получить доступ к полям класса A, но к какой же копии полей при этом он получит доступ – к копии класса B или класса C?

Чтобы избежать подобных проблем в языках Java и C# используется наследование только от одного класса. Почему же при этом возможно наследование от нескольких интерфейсов?

Наследование от интерфейсов отличается от наследования классов. При наследовании классов класс-наследник получает все реализованные методы от базового класса. При наследовании от интерфейсов класс-наследник обязуется реализовать методы, объявленные в интерфейсе. Поэтому в некоторых объектно-ориентированных языках программирования интерфейсы называют контрактами, а класс-наследник обязуется выполнить контракт по реализации нужной функциональности.

Также в некоторых объектно-ориентированных языках программирования интерфейсы называют примесями. Предполагается, что при наследовании есть основная функциональность, наследуемая от базового класса и вспомогательная функциональность, наследуемая от примесей.

При проектировании больших программных систем интерфейсам отводится несколько иная роль. В этом случае они рассматриваются не просто как средство нижнего уровня для устранения ромбовидного наследования, но как высокоуровневое средство, позволяющее независимо проектировать отдельные подсистемы. Производится декомпозиция системы на подсистемы, а интерфейсы определяют контракты взаимодействия между различными подсистемами, после чего проектирование подсистем может проводиться независимо.

Пример объявления интерфейса:

```
interface I1
{
    string I1_method();
}
```

В интерфейсе I1 объявлен метод I1_method, который не принимает параметров и возвращает строковое значение.

Данный метод можно рассматривать как аналог абстрактного метода.

Область видимости методов в интерфейсе не указывается, она определяется в наследуемом классе, реализующем методы.

Имена интерфейсов в языке C# принято начинать с большой латинской буквы I (сокращение от Interface), что позволяет в цепочке наследования отличать имена интерфейсов от имен классов.

Интерфейсы могут наследоваться от интерфейсов.

Пример наследования интерфейса от интерфейса:

```
interface I2 : I1
{
    string I2_method();
}
```

В этом случае интерфейс I2 содержит объявление метода I2_method а также объявление метода I1_method, унаследованного от интерфейса I1.

3.11 Наследование классов от интерфейсов

Рассмотрим пример класса, который наследуется как от класса, так и от интерфейсов.

```
class ExtendedClass2 : ExtendedClass1, I1, I2
{
    //В конструкторе вызывается конструктор базового класса
    public ExtendedClass2(int pi, int pi2, int pi3)
           : base(pi, pi2, pi3) {}

    //Реализация методов, объявленных в интерфейсах
    public string I1_method() { return ToString(); }
    public string I2_method() { return ToString(); }
}
```

В языке C# по аналогии с языком C++ для наследования как от классов так и от интерфейсов используется двоеточие. Базовый класс и интерфейсы перечисляются через запятую. В Java для наследования от классов используется ключевое слово extends, а для наследования от интерфейсов ключевое слово implements. Аналогичный код на Java выглядел бы следующим образом «class ExtendedClass2 extends ExtendedClass1 implements I1, I2».

Обратите внимание, что для объявления методов, унаследованных от интерфейса, не используется ключевое слово override. Методы,

унаследованные от интерфейса, не виртуальные, они как бы собственные методы класса.

Как и в случае наследования от абстрактных классов Visual Studio позволяет автоматически генерировать заглушки для методов интерфейсов. Создадим класс ClassForI1, наследуемый от интерфейса I1. В этом случае при нажатии правой кнопки мыши на имени интерфейса в контекстном меню появляется пункт автоматической реализации интерфейса (рис 11).

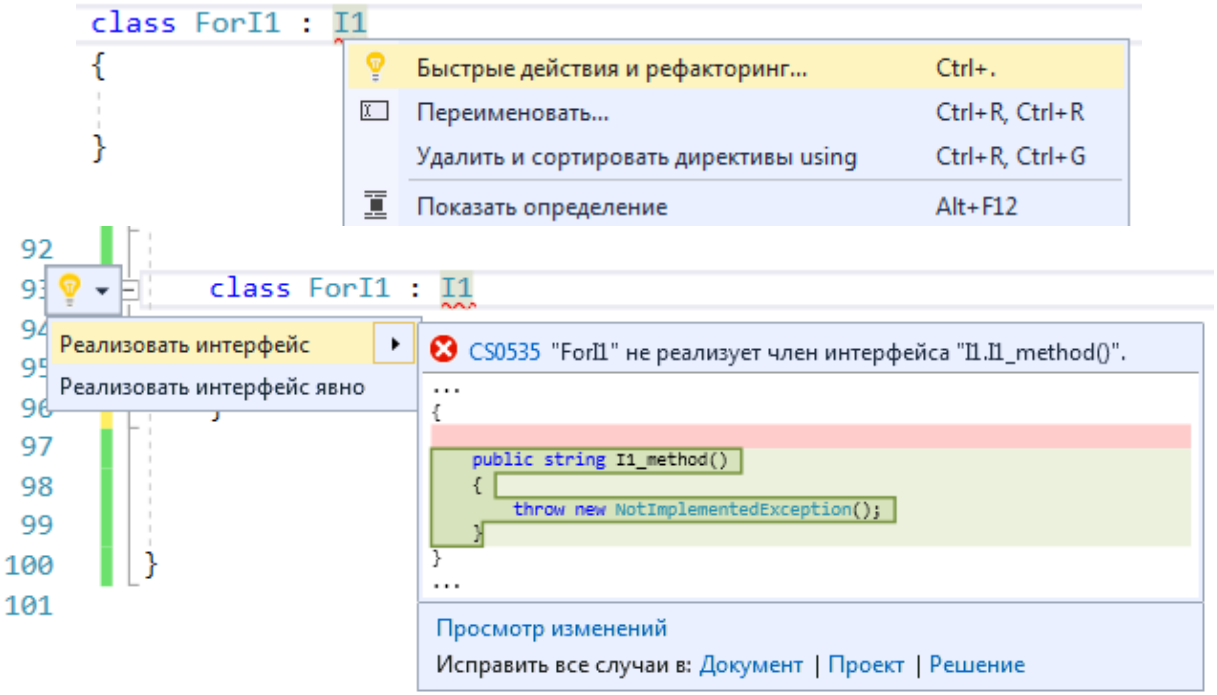

Рис. 9. Реализация интерфейса.

Отличие от абстрактного класса состоит в том, что в случае наследования от интерфейса подменю содержит два пункта «реализовать интерфейс» и «реализовать интерфейс явно». В случае выбора обоих пунктов будет сгенерирован следующий код, комментарий перед методом соответствует пункту меню:

```
class ClassForI1 : I1
{
    // реализовать интерфейс
    public string I1_method()
    {
```

```
        throw new NotImplementedException();
    }

    // реализовать интерфейс явно
    string I1.I1_method()
    {
        throw new NotImplementedException();
    }
}
```

Возникает вопрос, чем реализация интерфейса отличается от явной реализации, когда вызывается каждый из методов?

Можно сформулировать данный вопрос в виде практического примера. Дана следующая реализация класса:

```
class ClassForI1 : I1
{
    // реализовать интерфейс
    public string I1_method()
    {
        return "1";
    }

    // явно реализовать интерфейс
    string I1.I1_method()
    {
        return "2";
    }
}
```

Каким образом, используя данную реализацию, можно вывести в консоль число «12»?

Ответ на данный вопрос приведен в виде следующего фрагмента кода:

```
ClassForI1 c1 = new ClassForI1();
string str1 = c1.I1_method();
I1 i1 = (I1)c1;
string str2 = i1.I1_method();
Console.WriteLine(str1 + str2);
```

Рассмотрим данный код более подробно:

Сначала создается объект класса ClassForI1:

```
ClassForI1 c1 = new ClassForI1();
```

Через объект класса c1 вызывается метод I1_method, соответствующий пункту «реализовать интерфейс». Метод возвращает «1»:

```
string str1 = c1.I1_method();
```

Производится приведение переменной класса c1 к интерфейсному типу I1, для того чтобы вызвать метод явной реализации интерфейса:

```
I1 i1 = (I1)c1;
```

Затем через объект интерфейсного типа i1 вызывается метод I1.I1_method, соответствующий пункту «реализовать интерфейс явно». Метод возвращает «2»:

```
string str2 = i1.I1_method();
```

Таким образом, чтобы вызвать метод, соответствующий пункту «реализовать интерфейс явно», необходимо привести объект класса к интерфейсному типу.

3.12 Методы расширения

В языке C# существует уникальный механизм, который позволяет добавлять новые методы к уже реализованным классам, в том числе классам стандартной библиотеки. Сами разработчики .NET активно его используют, многие методы стандартных библиотек реализованы как методы расширения.

Причем методы стандартной библиотеки могут находиться в одной сборке (файле .dll), а методы расширения – в другой сборке. Важно чтобы они принадлежали к одному пространству имен.

Предположим, что нужно расширить класс ExtendedClass2 новым методом, однако по каким-либо причинам невозможно внести изменения в исходный код класса.

Тогда можно создать метод расширения следующим образом:

```
static class ExtendedClass2Extension
{
  public static int ExtendedClass2NewMethod(this ExtendedClass2 ec2, int i)
```

```
  {
    return i + 1;
  }
}
```

Данный класс является обычным классом, однако у него есть некоторые особенности. Метод расширения должен быть объявлен в статическом классе и должен быть статическим методом. Первый параметр метода расширения – объект расширяемого класса, перед которым указывается ключевое слово this.

Если откомпилировать класс ExtendedClass2Extension, то IntelliSense для класса ExtendedClass2 будет работать так, как показано на рис. 12.

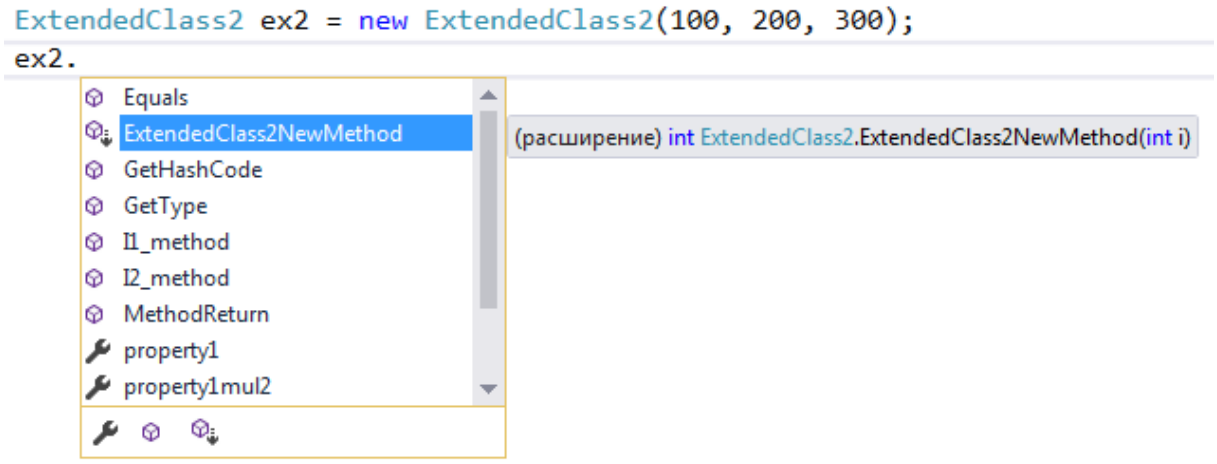

Рис. 10. Работа IntelliSense для метода расширения.

В этом случае метод расширения обозначен как обычный метод, но с дополнительной стрелкой, указывающей на то, что это метод расширения.

Таким образом, разработчику «кажется», что у класса ExtendedClass2 появился новый метод ExtendedClass2NewMethod, однако, данный метод объявлен в отдельном классе и возможно даже в отдельной сборке. Для успешной реализации метода расширения, пространства имен у базового класса и класса, содержащего метод расширения, должны совпадать.

3.13 Частичные классы

Еще один уникальный механизм, существующий в языке C#, это механизм частичных классов. Этот механизм позволяет объявить класс в нескольких файлах.

В языке C# этот механизм используется вместе со средствами автоматической генерации кода. В Visual Studio существует много средств, которые автоматически генерируют код, например для обращения к базе данных используется технология Entity Framework.

Если класс автоматически генерируется с помощью какого-либо средства, то он по умолчанию создается как частичный с помощью ключевого слова partial. Это позволяет создать другую «часть» данного класса в отдельном файле и вручную дописать необходимые методы к автоматически сгенерированному классу.

Казалось бы что эта возможность есть в языке C++, ведь в нем предусмотрено разделение класса на заголовочный файл и реализацию. Но в языке C++ невозможно разделить заголовочный файл на несколько файлов. В Java это принципиально невозможно, поскольку действует правило один класс – один файл. Поэтому, когда в Java возникла необходимость создания заглушек для сетевого взаимодействия, то для этого создали специальный шаблон проектирования из нескольких классов.

Пример объявления частичного класса. Файл «PartialClass1.cs»:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Classes
{
    partial class PartialClass
    {
        int i1;
```

```
        public PartialClass(int pi1, int pi2) { i1 = pi1; i2 = pi2;}

        public int MethodPart1(int i1, int i2)
        {
            return i1 + i2;
        }
    }
}
```

Данная часть класса содержит закрытую переменную класса, метод MethodPart1 и конструктор.

Пример объявления частичного класса. Файл «PartialClass2.cs»:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Classes
{
    partial class PartialClass
    {
        int i2;

        public override string ToString()
        {
            return "Частичный класс. i1=" + i1.ToString()
                                  + " i2=" + i2.ToString();
        }

        public string MethodPart2(string i1, string i2)
        {
            return i1 + i2;
        }
    }
}
```

Данная часть класса содержит закрытую переменную класса, метод MethodPart2 и переопределение виртуального метода ToString.

Работа механизма IntelliSense для частичного класса показана на рис 13.

Компилятор успешно соединил части частичного класса в нескольких файлах. Методы MethodPart1 и MethodPart2, объявленные в разных файлах, показаны в едином откомпилированном классе.

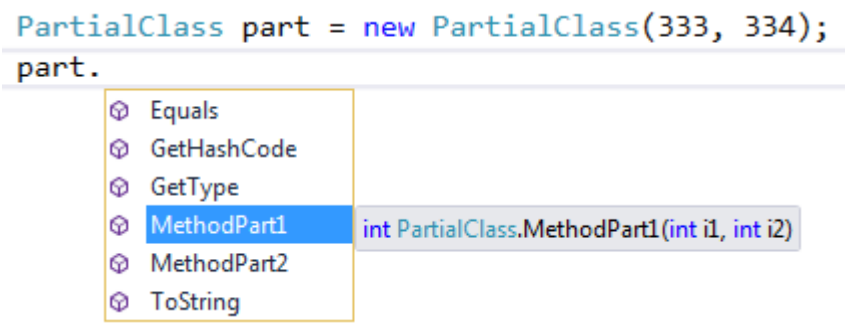

Рис. 11. Работа IntelliSense для частичного класса.

3.14 Создание диаграммы классов в Visual Studio

Для удобства разработки в Visual Studio существует возможность создания диаграммы классов.

Для добавления диаграммы классов необходимо добавить в проект элемент «диаграмма классов», как показано на рис. 14, 15.

Нужно нажать правую кнопку мыши на проекте в дереве проектов и выбрать в контекстном меню пункты «Добавить/Создать элемент».

Затем в диалоге добавления нового элемента следует выбрать пункт меню «диаграмма классов» (файл с расширением .cd).

Схема классов открывается в виде белого «холста», на который нужно переносить файлы из обозревателя решений. При этом все классы данного файла автоматически попадают на диаграмму. Пример такой диаграммы приведен на рис. 16.

Нотация диаграмм классов в Visual Studio практически полностью соответствует нотации диаграмм классов UML (Unified Modelling Language). Наследование классов показано незаштрихованной треугольной стрелкой, реализация интерфейсов – стрелкой в виде незаштрихованной окружности.

При нажатии на холсте правой кнопки мыши предоставляется возможность выбрать в контекстном меню пункт «экспорт схемы как изображения». В этом случае вся диаграмма классов экспортируется как

изображение в графическом формате (jpeg, png, другие форматы), что удобно для оформления документации.

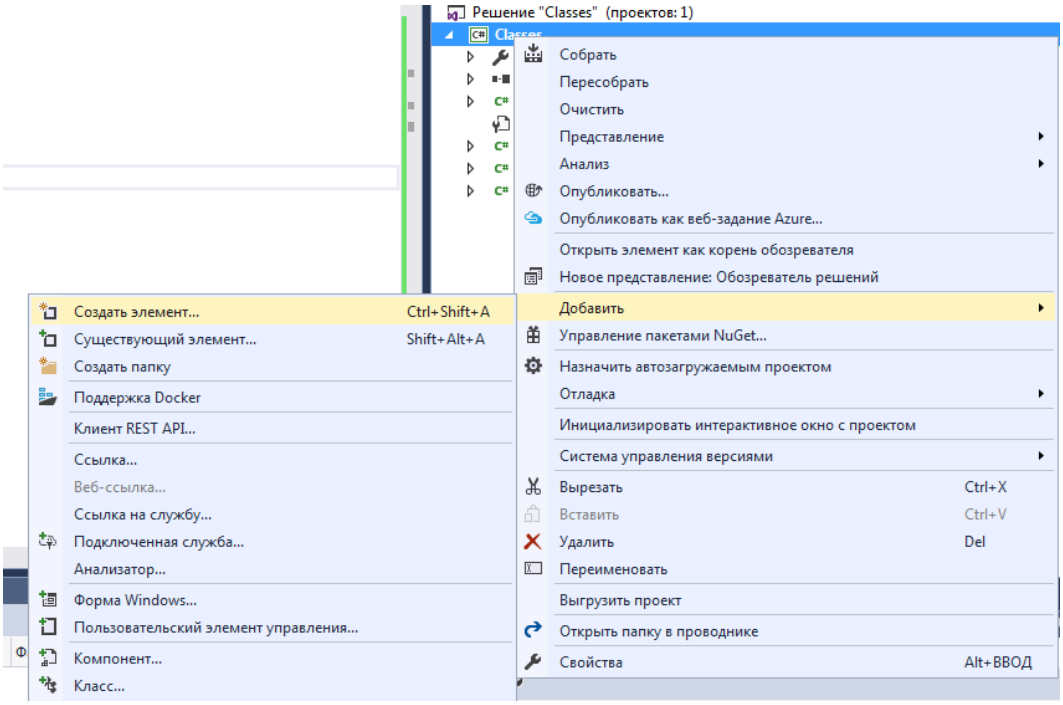

Рис. 12. Добавление диаграммы классов (шаг 1).

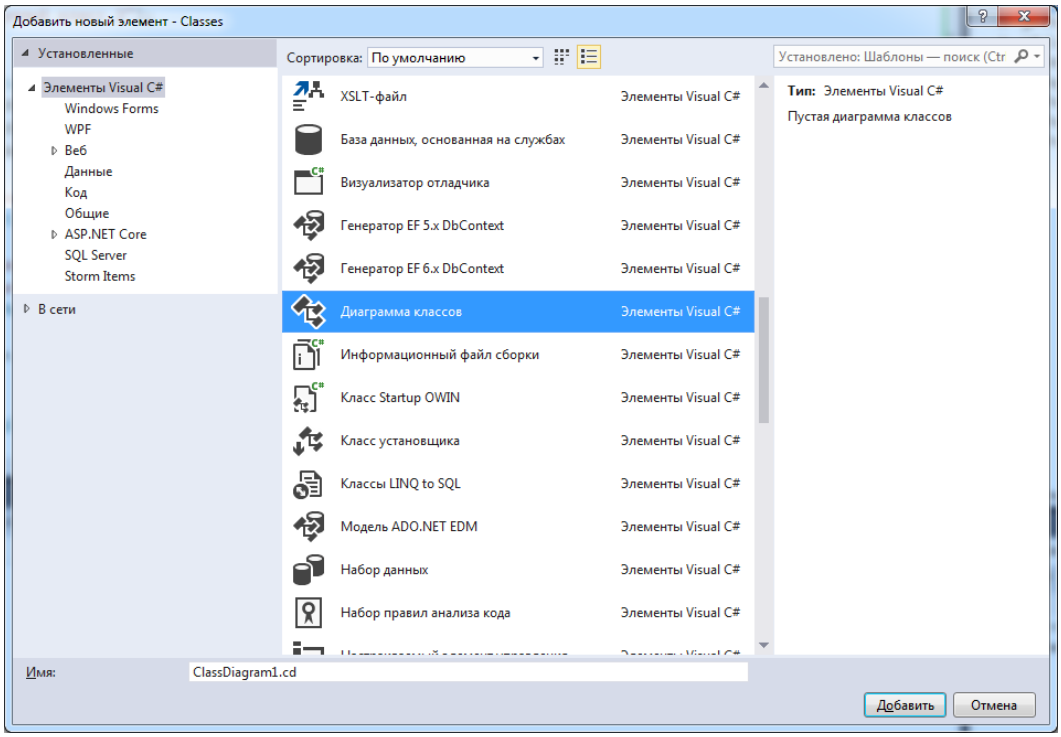

Рис. 13. Добавление диаграммы классов (шаг 2).

В панели кнопок диаграммы классов предусмотрены различные варианты отображения информации о полях классов (рис. 17):

- только имена полей;

- имена и типы полей;
- полная сигнатура.

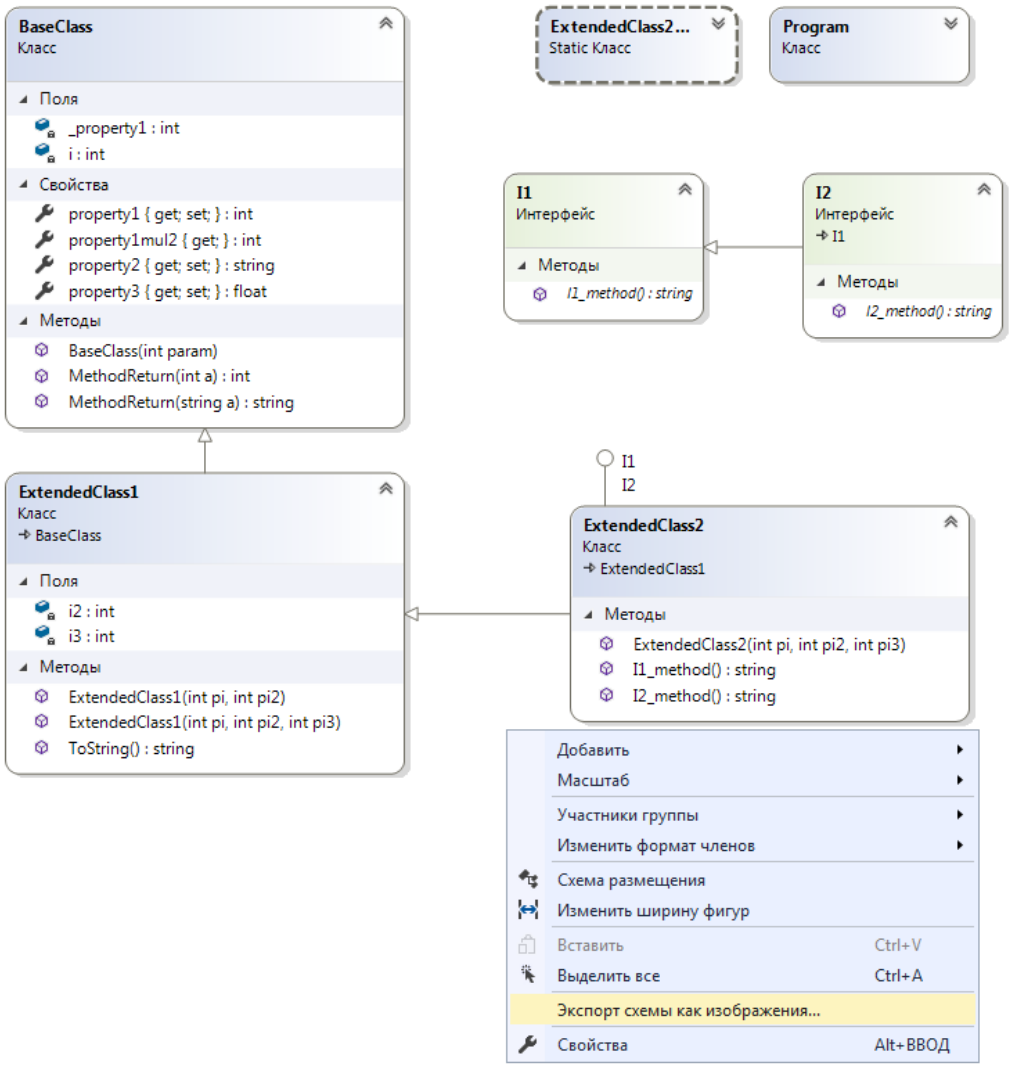

Рис. 14. Фрагмент диаграммы классов.

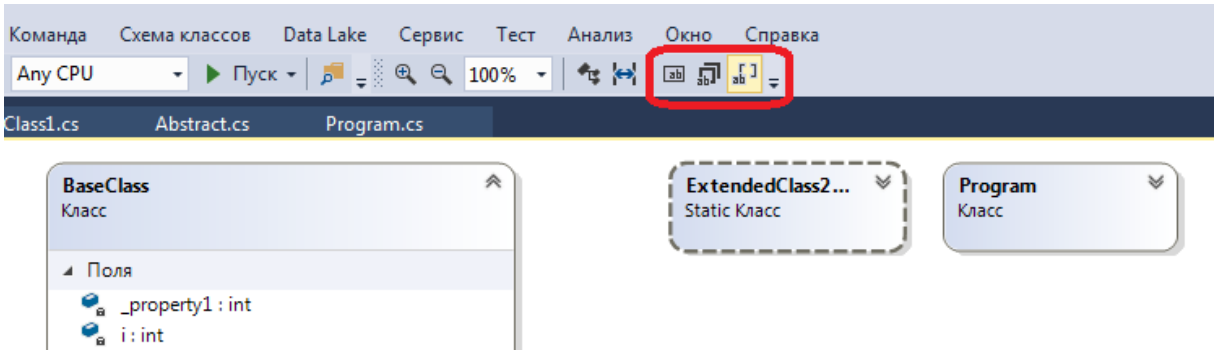

Рис. 15. Кнопки отображения информации о полях классов.

4 Задание. Проектирование и разработка системы классов «геометрические фигуры»

Необходимо спроектировать и разработать на языке C# систему классов геометрических фигур.

1. Программа должна быть разработана в виде консольного приложения на языке C#.
2. Абстрактный класс «Геометрическая фигура» содержит виртуальный метод для вычисления площади фигуры.
3. Класс «Прямоугольник» наследуется от «Геометрическая фигура». Ширина и высота объявляются как свойства (property). Класс должен содержать конструктор, принимающий параметры «ширина» и «высота».
4. Класс «Квадрат» наследуется от «Прямоугольник». Класс должен содержать конструктор по длине стороны.
5. Класс «Круг» наследуется от «Геометрическая фигура». Радиус объявляется как свойство (property). Класс должен содержать конструктор по параметру «радиус».
6. Для классов «Прямоугольник», «Квадрат», «Круг» необходимо переопределить виртуальный метод Object.ToString(), который возвращает в виде строки основные параметры фигуры и ее площадь.
7. Необходимо разработать интерфейс IPrint. Интерфейс содержит метод Print(), который не принимает параметров и возвращает void.
8. Для классов «Прямоугольник», «Квадрат», «Круг» необходимо реализовать наследование от интерфейса IPrint. Переопределяемый метод Print() выводит на консоль информацию, возвращаемую переопределенным методом ToString().

5 Пример выполнения задания

В качестве обобщающего примера в данном разделе приведены фрагменты **примера Figures_1** – программы, реализующей работу с геометрическими фигурами.

5.1 Абстрактный класс «Геометрическая фигура»

Основа системы классов для работы с геометрическими фигурами – абстрактный класс «Геометрическая фигура»:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Figures
{
    /// <summary>
    /// Класс фигура
    /// </summary>
    abstract class Figure
    {
        /// <summary>
        /// Тип фигуры
        /// </summary>
        public string Type
        {
            get
            {
                return this._Type;
            }
            protected set
            {
                this._Type = value;
            }
        }
        string _Type;

        /// <summary>
        /// Вычисление площади
        /// </summary>
        public abstract double Area();
        /// <summary>
        /// Приведение к строке, переопределение метода Object
        /// </summary>
        public override string ToString()
```

```
        {
            return this.Type + " площадью " +
this.Area().ToString();
        }
    }
}
```

Класс объявлен как абстрактный с помощью ключевого слова abstract.

Далее объявляется строковое свойство (property) Type, содержащее строковое наименование фигуры. Данной свойство объявлено в полной форме, хотя оно содержит стандартный код и могло бы быть объявлено как автоопределяемое свойство:

```
public string Type { get; protected set; }
```

Set-аксессор свойства объявлен с областью видимости protected, то есть присваивать значение данному свойству можно только в текущем классе и в классах-наследниках.

Далее объявляется абстрактный метод вычисления площади «double Area()», который должен быть определен в классах-наследниках. Он возвращает значение площади (тип double) и не принимает параметров, так как площадь должна вычисляться на основе внутренних данных классов геометрических фигур.

Затем переопределяется виртуальный метод ToString из класса Object для приведения к строковому типу.

5.2 Интерфейс IPrint

Интерфейс IPrint предназначен для вывода информации о геометрической фигуре:

```
namespace Figures
{
    interface IPrint
    {
        void Print();
    }
}
```

Интерфейс содержит метод Print(), который не принимает параметров и возвращает void.

В дальнейшем классы Прямоугольник, Квадрат и Круг будут реализовывать интерфейс IPrint. Переопределяемый метод Print() будет выводить в консоль информацию, возвращаемую переопределенным методом ToString().

5.3 Класс «Прямоугольник»

Класс «Прямоугольник» наследуется от абстрактного класса «Геометрическая фигура» и реализует интерфейс IPrint:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Figures
{
    class Rectangle : Figure, IPrint
    {
        /// <summary>
        /// Высота
        /// </summary>
        double height;

        /// <summary>
        /// Ширина
        /// </summary>
        double width;

        /// <summary>
        /// Основной конструктор
        /// </summary>
        /// <param name="ph">Высота</param>
        /// <param name="pw">Ширина</param>
        public Rectangle(double ph, double pw)
        {
            this.height = ph;
            this.width = pw;
            this.Type = "Прямоугольник";
        }

        /// <summary>
        /// Вычисление площади
```

```
        /// </summary>
        public override double Area()
        {
            double Result = this.width * this.height;
            return Result;
        }

        public void Print()
        {
            Console.WriteLine(this.ToString());
        }
    }
}
```

Класс содержит поля данных для высоты и ширины. Так как данные поля не доступны снаружи класса, они объявлены в виде обычных полей, а не в виде свойств.

Класс включает в себя конструктор, присваивающий значение ширины и высоты полям данных класса. Также инициализируется свойство Type, унаследованное от абстрактного класса.

Далее объявлен унаследованной от абстрактного класса Геометрическая фигура метод Area. Поскольку данный метод был объявлен как абстрактный, то он автоматически является виртуальным и должен быть переопределен с ключевым словом override.

Затем объявлен метод Print, унаследованный от интерфейса IPrint. Данный метод выводит в консоль результат работы метода ToString. Поскольку метод ToString не переопределялся в текущем классе, то его реализация берется из родительского класса, которым в данном случае является абстрактный класс Геометрическая фигура.

5.4 Класс «Квадрат»

Класс Квадрат наследуется от класса Прямоугольник и реализует интерфейс IPrint:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace Figures
{
    class Square : Rectangle, IPrint
    {
        public Square(double size)
            : base(size, size)
        {
            this.Type = "Квадрат";
        }
    }
}
```

Данный класс использует геометрическое определение квадрата, как прямоугольника с одинаковыми сторонами.

В этом классе фактически реализован только конструктор, принимающий сторону квадрата в качестве параметра. В начале работы конструктор класса Квадрат вызывает конструктор базового класса «base(size, size)», которым в данном случае является класс Прямоугольник. Сторона квадрата передается как длина и ширина прямоугольника в конструктор базового класса. В теле конструктора инициализируется только свойство Type.

Несмотря на то, что класс реализует интерфейс IPrint, в нем не содержится метода Print, так как этот метод унаследован от родительского класса.

5.5 Класс «Круг»

Класс Круг наследуется от абстрактного класса Геометрическая фигура и реализует интерфейс IPrint:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Figures
{
    class Circle : Figure, IPrint
    {
        /// <summary>
```

```
        /// Ширина
        /// </summary>
        double radius;

        /// <summary>
        /// Основной конструктор
        /// </summary>
        /// <param name="ph">Высота</param>
        /// <param name="pw">Ширина</param>
        public Circle(double pr)
        {
            this.radius = pr;
            this.Type = "Круг";
        }

        public override double Area()
        {
            double Result = Math.PI * this.radius * this.radius;
            return Result;
        }

        public void Print()
        {
            Console.WriteLine(this.ToString());
        }
    }
}
```

Реализация данного класса очень похожа на реализацию класса Прямоугольник, но в качестве полей данных хранится только радиус окружности, а площадь определяется по формуле площади круга.

5.6 Основная программа

Основная программа содержит примеры создания объектов классов и вывод информации о них с помощью метода Print, унаследованного от интерфейса IPrint:

```
using System;
using System.Linq;
using System.Text;

namespace Figures
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Rectangle rect = new Rectangle(5, 4);
            Square square = new Square(5);
            Circle circle = new Circle(5);

            rect.Print();
            square.Print();
            circle.Print();

            Console.ReadLine();
        }
    }
}
```

В результате выполнения программы в консоль будет выведено:

```
Прямоугольник площадью 20
Квадрат площадью 25
Круг площадью 78,5398163397448
```

5.7 Реализация с использованием современных возможностей C#

Более современный код на C# приведен в **примере Figures_2**.

5.8 Контрольные вопросы к разделу 4

1. Как объявить конструктор класса в языке C#?
2. Как объявить метод класса в языке C#?
3. Какие области видимости существуют в языке C#?
4. Что такое свойства и для чего они используются?
5. Что такое опорная переменная свойства?
6. Как используются get-аксессор и set-аксессор свойства?
7. Как задаются области видимости для свойств и аксессоров?
8. Приведите пример задания автоопределяемого свойства.
9. Приведите пример задания вычисляемого свойства.
10. Как объявить деструктор класса в языке C#?

11. Как объявить статические элементы класса в языке C#?
12. Как используется конструкция using static в языке C#?
13. Как реализуется наследование класса от класса?
14. Как из конструктора класса вызвать конструктор базового класса?
15. Как из конструктора класса вызвать другой конструктор текущего класса?
16. Как переопределить виртуальный метод?
17. В чем разница между ключевыми словами override и new при переопределении виртуального метода?
18. В чем сходства и различия между виртуальными и абстрактными методами?
19. Что такое абстрактный класс?
20. Как в Visual Studio сгенерировать заглушки методов при наследовании от абстрактного класса?
21. Что такое интерфейсы и для чего они используются в языке C#?
22. В чем сходства и различия между интерфейсами и абстрактными классами?
23. Можно ли в языке C# унаследовать класс от нескольких классов? От нескольких интерфейсов? Почему?
24. Как реализуется наследование класса от класса и интерфейсов?
25. Являются ли методы, унаследованные от интерфейсов, виртуальными?
26. Как в Visual Studio сгенерировать заглушки методов при наследовании от интерфейса?
27. В чем разница между «реализацией интерфейса» и «явной реализацией интерфейса»?
28. Что такое методы расширения и как они реализуются в языке C#?

29. Что такое частичные классы и как они реализуются в языке C#?
30. Как создать диаграмму классов в Visual Studio?