

Московский государственный технический университет  
имени Н.Э. Баумана

---

Факультет «Информатика и системы управления»  
Кафедра «Системы обработки информации и управления»

**Э.Н. Самохвалов, Ю.Е. Гапанюк**

**ВВЕДЕНИЕ В ПРОЕКТИРОВАНИЕ  
И РАЗРАБОТКУ  
ИНТЕРНЕТ-ПРИЛОЖЕНИЙ  
Часть I**

Электронное учебное издание

*Учебное пособие по дисциплине  
«Базовые компоненты интернет-технологий»*

Москва

(С) 2015 МГТУ им. Н.Э. БАУМАНА

**Самохвалов Э.Н., Гапанюк Ю.Е.**

Введение в проектирование и разработку интернет-приложений. Часть I. Электронное учебное издание. – М.: МГТУ имени Н.Э. Баумана, 2015. 249 с.

Издание предназначено для ознакомления с основами языка программирования C#. Рассматриваются основы среды исполнения .NET, основные конструкции языка C#, основы объектно-ориентированного программирования в C#, основы работы с коллекциями, файловой системой, рефлексией, основы параллельной обработки данных, введение в технологию разработки оконных пользовательских интерфейсов Windows Forms.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по специальности «Информатика и вычислительная техника».

*Электронное учебное издание*

**Самохвалов Эдуард Николаевич**

**Гапанюк Юрий Евгеньевич**

**ВВЕДЕНИЕ В ПРОЕКТИРОВАНИЕ И РАЗРАБОТКУ  
ИНТЕРНЕТ-ПРИЛОЖЕНИЙ. ЧАСТЬ I**

## ОГЛАВЛЕНИЕ

<b>ПРЕДИСЛОВИЕ</b> .....	<b>7</b>
<b>ВВЕДЕНИЕ</b> .....	<b>8</b>
<b>1 КРАТКАЯ ХАРАКТЕРИСТИКА ЯЗЫКА ПРОГРАММИРОВАНИЯ C#</b> .....	<b>11</b>
<b>2 СРЕДА ИСПОЛНЕНИЯ .NET</b> .....	<b>13</b>
2.1 КРАТКОЕ ОПИСАНИЕ СРЕДЫ ИСПОЛНЕНИЯ .NET .....	13
2.2 ОСОБЕННОСТЬ КОМПИЛЯЦИИ И ИСПОЛНЕНИЯ ПРИЛОЖЕНИЙ ДЛЯ СРЕДЫ ИСПОЛНЕНИЯ .NET	15
<b>3 ОСНОВЫ ЯЗЫКА C#</b> .....	<b>17</b>
3.1 ОРГАНИЗАЦИЯ ТИПОВ ДАННЫХ В C#.....	17
3.2 БАЗОВЫЕ ТИПЫ ДАННЫХ .....	19
3.3 ПРЕОБРАЗОВАНИЯ ТИПОВ .....	21
3.4 ИСПОЛЬЗОВАНИЕ МАССИВОВ .....	23
3.5 КОНСОЛЬНЫЙ ВВОД/ВЫВОД.....	24
3.6 ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА ПРОГРАММИРОВАНИЯ C# .....	25
3.6.1 <i>Пространства имен и сборки</i> .....	30
3.6.2 <i>Условные операторы</i> .....	34
3.6.3 <i>Операторы цикла</i> .....	35
3.6.4 <i>Обработка исключений</i> .....	36
3.6.5 <i>Вызов методов, передача параметров и возврат значений</i> .....	39
3.7 XML-КОММЕНТАРИИ.....	42
3.8 ДИРЕКТИВЫ ПРЕПРОЦЕССОРА.....	43
3.9 ПРИМЕР КОНСОЛЬНОГО ВВОДА/ВЫВОДА С ПРЕОБРАЗОВАНИЕМ ТИПОВ ДАННЫХ .....	45
<b>4 ОСНОВЫ ООП В C#</b> .....	<b>47</b>
4.1 ОБЪЯВЛЕНИЕ КЛАССА И ЕГО ЭЛЕМЕНТОВ.....	48
4.1.1 <i>Объявление конструктора</i> .....	48
4.1.2 <i>Объявление методов</i> .....	49
4.1.3 <i>Объявление свойств</i> .....	49
4.1.4 <i>Объявление деструкторов</i> .....	53
4.1.5 <i>Объявление статических элементов класса</i> .....	54
4.2 НАСЛЕДОВАНИЕ КЛАССА ОТ КЛАССА .....	54
4.2.1 <i>Вызов конструкторов из конструкторов</i> .....	55
4.2.2 <i>Виртуальные методы</i> .....	56
4.3 АБСТРАКТНЫЕ КЛАССЫ И МЕТОДЫ.....	58
4.4 ИНТЕРФЕЙСЫ .....	60
4.5 НАСЛЕДОВАНИЕ КЛАССОВ ОТ ИНТЕРФЕЙСОВ.....	63
4.6 МЕТОДЫ РАСШИРЕНИЯ .....	66
4.7 ЧАСТИЧНЫЕ КЛАССЫ .....	68

4.8	СОЗДАНИЕ ДИАГРАММЫ КЛАССОВ В VISUAL STUDIO .....	70
4.9	ПРИМЕР КЛАССОВ ДЛЯ РАБОТЫ С ГЕОМЕТРИЧЕСКИМИ ФИГУРАМИ .....	73
4.9.1	<i>Абстрактный класс «Геометрическая фигура»</i> .....	73
4.9.2	<i>Интерфейс IPrint</i> .....	74
4.9.3	<i>Класс «Прямоугольник»</i> .....	75
4.9.4	<i>Класс «Квадрат»</i> .....	76
4.9.5	<i>Класс «Круг»</i> .....	77
4.9.6	<i>Основная программа</i> .....	78
<b>5</b>	<b>РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ООП В C#</b> .....	<b>79</b>
5.1	ПЕРЕЧИСЛЕНИЯ.....	79
5.2	ПЕРЕГРУЗКА ОПЕРАТОРОВ .....	81
5.3	ОБОБЩЕНИЯ .....	91
5.4	ДЕЛЕГАТЫ .....	95
5.5	ЛЯМБДА-ВЫРАЖЕНИЯ .....	99
5.6	ОБОБЩЕННЫЕ ДЕЛЕГАТЫ FUNC И ACTION .....	102
5.7	ГРУППОВЫЕ ДЕЛЕГАТЫ.....	106
5.8	СОБЫТИЯ.....	107
<b>6</b>	<b>РАБОТА С КОЛЛЕКЦИЯМИ</b> .....	<b>114</b>
6.1	СТАНДАРТНЫЕ КОЛЛЕКЦИИ.....	114
6.1.1	<i>Обобщенный список</i> .....	116
6.1.2	<i>Необобщенный список</i> .....	118
6.1.3	<i>Обобщенные стек и очередь</i> .....	119
6.1.4	<i>Обобщенный словарь</i> .....	122
6.1.5	<i>Кортеж</i> .....	124
6.1.6	<i>Сортировка коллекций</i> .....	125
6.2	СОЗДАНИЕ НЕСТАНДАРТНОЙ КОЛЛЕКЦИИ НА ОСНОВЕ СТАНДАРТНОЙ КОЛЛЕКЦИИ.....	130
6.3	СОЗДАНИЕ НЕСТАНДАРТНОЙ КОЛЛЕКЦИИ БЕЗ ИСПОЛЬЗОВАНИЯ СТАНДАРТНЫХ КОЛЛЕКЦИЙ 138	
6.3.1	<i>Классы простого списка</i> .....	138
6.3.2	<i>Класс стека</i> .....	146
<b>7</b>	<b>РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ</b> .....	<b>148</b>
7.1	ПОЛУЧЕНИЕ ДАННЫХ О ФАЙЛАХ И КАТАЛОГАХ.....	148
7.2	ЧТЕНИЕ И ЗАПИСЬ ТЕКСТОВЫХ ФАЙЛОВ.....	150
7.3	СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ ОБЪЕКТОВ.....	154
7.3.1	<i>Бинарная сериализация/десериализация</i> .....	155
7.3.2	<i>Сериализация/десериализация в формат XML</i> .....	158
<b>8</b>	<b>РЕФЛЕКСИЯ</b> .....	<b>162</b>
8.1	РАБОТА СО СБОРКАМИ .....	163

8.2	РАБОТА С ТИПАМИ ДАННЫХ .....	164
8.3	ДИНАМИЧЕСКИЕ ДЕЙСТВИЯ С ОБЪЕКТАМИ КЛАССОВ .....	167
8.4	РАБОТА С АТТРИБУТАМИ .....	168
8.5	ИСПОЛЬЗОВАНИЕ РЕФЛЕКСИИ НА УРОВНЕ ОТКОМПИЛИРОВАННЫХ ИНСТРУКЦИЙ.....	173
8.6	САМООТБРАЖАЕМОСТЬ .....	176
<b>9</b>	<b>ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ .....</b>	<b>178</b>
9.1	ИСПОЛЬЗОВАНИЕ КЛАССА THREAD .....	179
9.2	ИСПОЛЬЗОВАНИЕ КЛАССА TASK.....	182
9.3	ВОЗВРАТ РЕЗУЛЬТАТА ВЫПОЛНЕНИЯ ПОТОКА С ИСПОЛЬЗОВАНИЕМ КЛАССА TASK .....	184
9.4	ИСПОЛЬЗОВАНИЕ КОНСТРУКЦИЙ ASYNC И AWAIT.....	192
<b>10</b>	<b>РЕАЛИЗАЦИЯ АЛГОРИТМА ПОИСКА С ОПЕЧАТКАМИ .....</b>	<b>194</b>
10.1	РАССТОЯНИЕ ДАМЕРАУ-ЛЕВЕНШТЕЙНА .....	194
10.2	ВЫЧИСЛЕНИЕ РАССТОЯНИЯ ДАМЕРАУ-ЛЕВЕНШТЕЙНА .....	196
10.3	ПРИМЕР ВЫЧИСЛЕНИЯ РАССТОЯНИЯ ДАМЕРАУ-ЛЕВЕНШТЕЙНА .....	200
10.4	АЛГОРИТМ ВАГНЕРА-ФИШЕРА ВЫЧИСЛЕНИЯ РАССТОЯНИЯ ДАМЕРАУ-ЛЕВЕНШТЕЙНА .	201
<b>11</b>	<b>ОСНОВЫ РАЗРАБОТКИ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ WINDOWS FORMS.....</b>	<b>204</b>
11.1	СОЗДАНИЕ ПРОЕКТА .....	204
11.2	ПРИМЕР РАБОТЫ С КНОПКОЙ И ТЕКСТОВЫМ ПОЛЕМ .....	206
11.3	ПРИМЕР РАБОТЫ С ТАЙМЕРОМ.....	212
11.4	ПРИМЕР ОТКРЫТИЯ ДОЧЕРНИХ ОКОН.....	216
<b>12</b>	<b>ПРИМЕР МНОГОПОТОЧНОГО ПОИСКА В ТЕКСТОВОМ ФАЙЛЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ WINDOWS FORMS.....</b>	<b>223</b>
12.1	ЧТЕНИЕ ИНФОРМАЦИИ ИЗ ТЕКСТОВОГО ФАЙЛА .....	224
12.2	ЧЕТКИЙ ПОИСК В ТЕКСТОВОМ ФАЙЛЕ.....	227
12.3	НЕЧЕТКИЙ ПОИСК В ТЕКСТОВОМ ФАЙЛЕ .....	229
12.4	ФОРМИРОВАНИЕ ОТЧЕТА .....	236
	<b>КОНТРОЛЬНЫЕ ВОПРОСЫ.....</b>	<b>241</b>
	К РАЗДЕЛУ 2 .....	241
	К РАЗДЕЛУ 3 .....	241
	К РАЗДЕЛУ 4 .....	242
	К РАЗДЕЛУ 5 .....	244
	К РАЗДЕЛУ 6 .....	244
	К РАЗДЕЛУ 7 .....	245
	К РАЗДЕЛУ 8 .....	245
	К РАЗДЕЛУ 9 .....	246
	К РАЗДЕЛУ 10 .....	246
	К РАЗДЕЛУ 11 .....	246

К РАЗДЕЛУ 12 .....	247
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>248</b>
<b>ИСТОЧНИКИ .....</b>	<b>249</b>

## **Предисловие**

Данное учебное пособие предназначено для ознакомления с основами языка программирования C#.

В пособии рассматриваются основы среды исполнения .NET, основные конструкции языка C#, основы объектно-ориентированного программирования в C#, основы работы с коллекциями, файловой системой, рефлексией, основы параллельной обработки данных, введение в технологию разработки оконных пользовательских интерфейсов Windows Forms.

Пособие использует принцип обучения на основе примеров. Все изучаемые технологии рассматриваются в виде последовательности примеров, архив с примерами предоставляется в виде приложения к пособию. Ссылки на номера примеров даны в тексте пособия.

Все программные продукты, рассматриваемые в пособии, являются свободно распространяемыми или предоставляются МГТУ имени Н.Э. Баумана по академической лицензии для использования в учебном процессе.

## Введение

Особенностью современного состояния дел в области разработки интернет-приложений является большое количество используемых технологий.

На стороне веб-браузера используются язык разметки HTML, технология каскадных таблиц стилей CSS, язык программирования JavaScript.

На стороне веб-сервера в настоящее время используется большое количество фреймворков (каркасов) для веб-разработки на различных языках программирования, наиболее известными из которых являются PHP, Python, Perl, C#, Java, JavaScript.

Поэтому использование какого-либо фреймворка предполагает знание соответствующего языка программирования. В частности использованием фреймворка ASP.NET MVC, который в настоящее время достаточно популярен в среде веб-разработчиков, предполагает знание языка программирования «C#».

Дисциплина «Базовые компоненты интернет-технологий» читается на кафедре «Системы обработки информации и управления» в третьем семестре. Основным предметом изучения в рамках данной дисциплины является язык программирования «C#». Это обусловлено следующими причинами:

- В первых двух семестрах студенты кафедры изучают языки программирования «C» и «C++». Поэтому к третьему семестру студенты знакомы с синтаксисом языка «C++» и основами объектно-ориентированного программирования на «C++». Язык программирования «C#» имеет много общих особенностей с языком «C++». Чтение языка «C#» непосредственно после



«С++» позволяет сконцентрироваться на сходствах и различиях между этими языками.

- В отличие от языка «С++», который считается классическим объектно-ориентированным языком программирования, язык «С#» содержит ряд конструкций, присущих объектно-ориентированным языкам следующего поколения, например таких конструкций как делегаты, рефлексия.
- Для языка «С#» реализовано большое количество прикладных технологий, которые используются при разработке промышленных приложений: Windows Forms и WPF для разработки оконного пользовательского интерфейса, LINQ и Entity Framework для обработки данных, ASP.NET MVC для разработки веб-приложений. Использование языка «С++» возможно только с ограниченным набором прикладных технологий (например, с Windows Forms), разработка веб-приложений на основе ASP.NET MVC не предполагает использования языка «С++».
- Дисциплина «Разработка интернет-приложений», которая читается на кафедре «Системы обработки информации и управления» в пятом семестре включает изучение технологии ASP.NET MVC. Поэтому знание языка «С#» является необходимым требованием для изучения данной дисциплины.

Цель преподавания дисциплины «Базовые компоненты интернет-технологий» состоит в содействии формированию у обучающихся следующих компетенций:

- способен разрабатывать и отлаживать компоненты аппаратно-программных комплексов с помощью современных автоматизированных средств проектирования (ПК-7);
- умеет разрабатывать интерфейсы «человек - ЭВМ» (ПК-12);

В результате изучения дисциплины «Базовые компоненты интернет-технологий»:

1. Студент должен знать:
  - основы языка программирования «С#».
2. Студент должен уметь:
  - разрабатывать консольные и оконные приложения с использованием языка программирования «С#»;
  - разрабатывать интерфейсы «человек-ЭВМ» с использованием технологии Windows Forms.
3. Студент должен иметь навыки:
  - разработки консольных и оконных приложений с использованием языка программирования «С#»;
  - разработки интерфейсов «человек-ЭВМ» с использованием технологии Windows Forms.

Данное учебное пособие в целом соответствует содержанию дисциплины «Базовые компоненты интернет-технологий», которая читается в третьем семестре на кафедре «Системы обработки информации и управления».

## **1 Краткая характеристика языка программирования C#**

Язык программирования C#, разработанный в компании Microsoft для платформы .NET, является современным объектно-ориентированным языком программирования общего назначения.

С помощью языка программирования C# возможна разработка консольных приложений, оконных приложений с использованием технологий Windows Forms и WPF, веб-приложений с использованием технологий ASP.NET и ASP.NET MVC. Для обработки данных предназначены такие технологии как LINQ и Entity Framework.

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

В 2005 году вышла версия 2.0. В нее были включены средства и библиотеки для разработки оконных приложений на основе технологии Windows Forms.

В 2008 году вышла версия 3.5. В нее были включены средства и библиотеки для разработки оконных приложений на основе технологии WPF. Также была добавлена технология LINQ, которая представляет собой SQL-подобный язык запросов, встроенный в C#. Этот новаторский подход Андерса Хейлсберга до сих пор не имеет развитых аналогов в других языках программирования. Фактически версия 3.5 сформировала основную концепцию языка, которая с тех пор не претерпела принципиальных изменений, хотя в новые версии языка добавлено

большое количество новых синтаксических конструкций и библиотек, особенно для параллельной и асинхронной обработки данных.

Язык C# продолжает активно развиваться. Из новых проектов хотелось бы отметить проект модульного компилятора Roslyn, который предоставляет разработчикам API для различных этапов компиляции, что облегчает создание новых языков программирования для платформы .NET.

В целом хочется отметить, что Андерсу Хейлсбергу удалось не только решить сложную техническую задачу по созданию быстрого компилятора, но главное создать концептуально удачный язык программирования, который подходит как для задач обучения, так и для практической разработки крупных проектов. Основными чертами языка программирования C# являются:

- синтаксис, в целом унаследованный от C++ и Java;
- некоторые конструкции языка, унаследованные из Паскаля, но «обернутые» в синтаксис подобный C++.
- очень хорошая читаемость кода, что очень важно при разработке больших проектов;
- технология LINQ, которая облегчает обработку сложных структур данных;
- большое количество библиотек для решения различных задач.

Конечно, в рамках небольшого пособия можно рассмотреть только основы языка программирования C#. Для углубленного изучения C# можно рекомендовать следующие книги:

1. Книга Кристиана Нейгела с соавторами [1] включает 1440 страниц, а также дополнительные главы на CD. Содержит удачное базовое описание языка C# (около 300 страниц) а также описание большого количества технологий в дополнительных главах.

2. Книга Герберта Шилдта [2] включает около 1000 страниц. В основном книга посвящена только C# и некоторым технологиям. Языковые особенности C# рассмотрены более глубоко, чем в книге Нейгела. Отметим, что Г. Шилдт написал книги по C++, C# и Java, поэтому он уделяет серьезное внимание сходству и различию между языками.
3. Учебник Т.А. Павловской [3] включает около 400 страниц. Этот учебник, несомненно, является удачной отечественной работой. Но, к сожалению, в данном учебнике рассматривается только C# 2.0 версии 2005 года.

Также хочется отметить, что среда исполнения .NET и язык программирования C# детально документированы в справочной системе Microsoft – MSDN (Microsoft Developer Network). Сайт MSDN – <https://msdn.microsoft.com>

Данное пособие опирается на примеры. Номера примеров указаны в соответствующих разделах. В тексте пособия приводятся фрагменты примеров и комментарии к ним. Читатель может открыть примеры с помощью средств разработки, предназначенных для языка C#:

- Microsoft Visual Studio – полная версия среды разработки доступна во многих ВУЗах по академической лицензии. Также существует свободно-распространяемая express-версия Visual Studio.
- MonoDevelop – свободно-распространяемая среда разработки для платформы Mono, кроссплатформенного аналога .NET.

## **2 Среда исполнения .NET**

### **2.1 Краткое описание среды исполнения .NET**

Среда исполнения .NET представляет собой виртуальную среду исполнения программ. Можно упрощенно сказать, что это программно-

реализованная модель микропроцессора, которая исполняет команды на специфическом машинном языке. Этот язык называется MSIL (Microsoft Intermediate Language) или иногда IL (Intermediate Language).

Это язык машинных команд, который подобен другим языкам машинных команд, например микропроцессора Intel.

Как для команд микропроцессора Intel существует язык ассемблера, так и для MSIL существует специфический язык ассемблера. Программу на этом языке можно получить путем дизассемблирования исполняемого файла .NET.

Основным компонентом .NET, который используется для выполнения MSIL-кода, является Common Language Runtime (CLR) или .NET runtime. Код, исполняемый под управлением CLR, иногда называют управляемым (managed) кодом.

Другой важнейшей составляющей платформы .NET является сборщик мусора (garbage collector). В программах на C++ память выделяется с помощью оператора `new` и освобождается с помощью оператора `delete`. В языках платформы .NET существует оператор выделения памяти (в C# он также называется `new`), но оператор освобождения памяти принципиально отсутствует. Среда .NET сама следит за тем, нужна ли программе выделенная память и если не нужна, то память автоматически освобождается сборщиком мусора. Поэтому в среде исполнения .NET невозможны ошибки утечки памяти по причине того, что разработчик не вызвал оператор `delete`.

Следует отметить, что среда исполнения .NET архитектурно очень похожа на среду исполнения Java и среда исполнения Java исторически появилась первой. В Java язык MSIL называется байт-кодом (byte code). На платформе Java также реализован сборщик мусора, в языке Java как и в C# существует оператор `new` и отсутствует оператор `delete`.

## 2.2 Особенность компиляции и исполнения приложений для среды исполнения .NET

Рассмотрим процесс компиляции и компоновки для языков без использования среды исполнения. Этот процесс приведен на рис. 1.

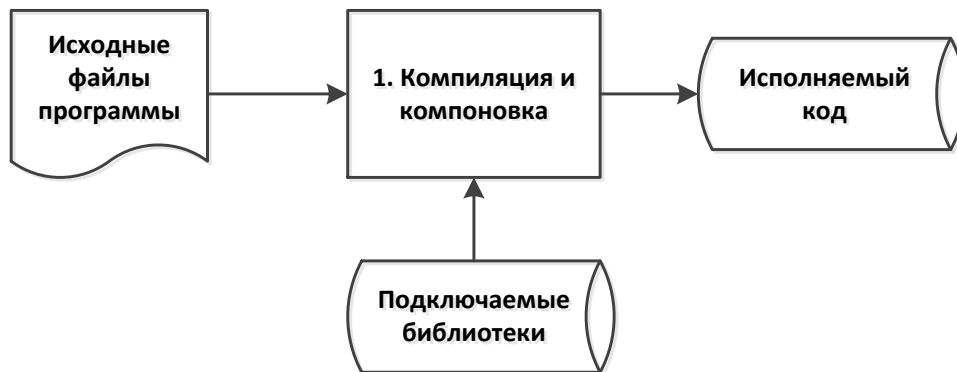


Рис. 1. Процесс компиляции и компоновки для языков без использования среды исполнения.

В этом случае исходные файлы программы сначала компилируются, затем скомпилированные модули компонуются, к ним присоединяются библиотеки и получается исполняемый код. Отметим, что исполняемый код представляет собой машинные команды для конкретного микропроцессора.

Безусловно, процесс на рис. 1 представлен схематично и может существенно различаться в деталях для конкретных языков и технологий.

Если считать компиляцию и компоновку частями одного процесса, то можно сказать что процесс компиляции и компоновки для языков без использования среды исполнения выполняется в одну фазу. В противоположность этому процесс компиляции и компоновки на платформе .NET выполняется в две фазы, что показано на рис. 2.

В этом случае в результате компиляции и компоновки формируется MSIL-код, так называемый файл сборки (.dll или .exe).

При запуске файла сборки происходит процесс, который называется JIT-компиляцией. JIT – это сокращение от Just In Time. JIT-компиляция – это процесс преобразования MSIL-кода в исполняемый код для

конкретного микропроцессора, который установлен на данном компьютере. JIT-компиляция выполняется в памяти, исполняемый код выполняется немедленно и не сохраняется на жесткий диск.

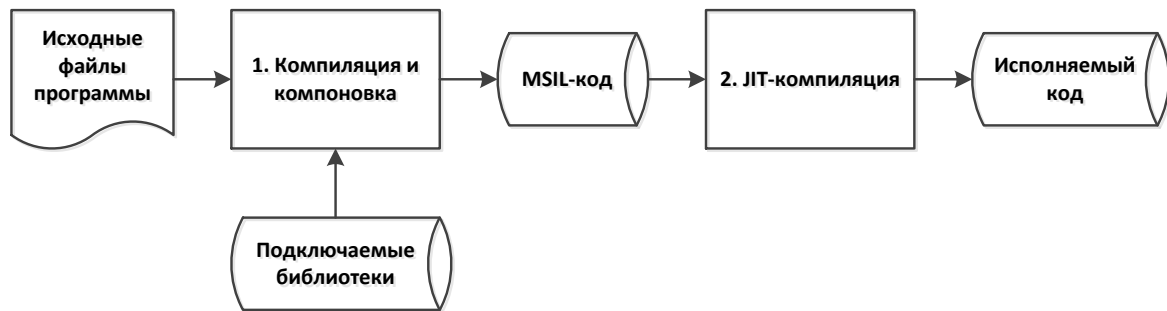


Рис. 2. Процесс компиляции, компоновки и исполнения программы для .NET-платформы.

Разработчику, который не знаком с виртуальными средами исполнения .NET или Java подобная схема может показаться на первый взгляд совершенно бессмысленной. Действительно, ведь в случае использования, например, C++ в результате компиляции и компоновки получается «быстрый» машинный код, который сразу исполняется. В случае использования виртуальной среды сначала генерируется «ненужный» промежуточный код, только затем он переводится в быстрый машинный код и выполняется. Вроде бы вторая схема в плане исполнения кода очевидно медленнее первой. Но парадокс в том, что вторая схема оказывается, как правило, быстрее. Для этого есть две причины.

Первая причина состоит в том, что для программы на C++, как правило, неизвестно на какой платформе (каком конкретно микропроцессоре) будет исполняться код. Поэтому компилятор обычно генерирует то подмножество машинных команд, которое будет выполняться на большинстве микропроцессоров, то есть приоритет при генерации кода отдается совместимости с микропроцессором в ущерб производительности. В случае виртуальной среды такой проблемы не возникает, JIT-компилятор может определить, какой микропроцессор



установлен на данном компьютере и может генерировать команды, которые оптимальны по производительности для данного микропроцессора.

Вторая причина состоит в том, что JIT-компилятор имеет встроенный профилировщик кода, то есть может оценить качество и скорость выполнения сгенерированного кода. В случае необходимости JIT-компилятор может повторно сгенерировать машинный код, который будет выполняться быстрее.

Следует отметить, что исторически JIT-компиляторы впервые появились на Java-платформе. В первых версиях Java исполняемый код (байт-код) интерпретировался, что приводило к значительному снижению производительности. Затем появились JIT-компиляторы, которые «на лету» преобразовывали байт-код в машинный код, что улучшило производительность Java-машины. В среде .NET JIT-компилятор существовал изначально, поэтому даже у первых версий платформы .NET практически никогда не существовало проблем с производительностью.

## **3 Основы языка C#**

### **3.1 Организация типов данных в C#**

Особенность типов данных C# заключается в том, что для всей среды исполнения .NET используется единая общая система типов – CTS (Common Type System). Эта система типов используется как в C#, так и в Visual Basic .NET и других языках .NET-платформы.

Все типы данных разделяются на две большие категории – типы-значения (value type) и ссылочные типы (reference type).

Типы-значения хранятся в области памяти, доступ к которой организован в виде стека. К типам-значениям относятся:

- базовые типы данных;
- перечисления (enum);

- структуры (struct).

Если переменная типа-значения передается как параметр в метод, то в стеке делается копия значения и передается в метод, исходное значение не изменяется. Такой порядок копирования значений в частности удобен для примитивных типов.

Ссылочные типы хранятся в динамически распределяемой области памяти, которую принято называть кучей (heap). Для работы с такой областью памяти используются ссылки, которые фактически являются указателями. К ссылочным типам относятся:

- классы;
- интерфейсы;
- массивы;
- делегаты.

Если переменная ссылочного типа передается как параметр в метод, то при этом не делается копия значения (как в случае типа-значения), а передается указатель на переменную, поэтому если переменная изменяется в методе, то исходное значение также изменяется. Такой порядок копирования значений в частности удобен для передачи объектов классов.

В отличие от C++ в C# использование типов-значений и ссылочных типов практически ничем не различается, в частности не используются специальные символы для обозначения ссылок «&». Использование указателей (которые как и в C++ обозначаются символом «\*») возможно в так называемом небезопасном (unsafe) режиме. Использование этого режима возможно для взаимодействия с аппаратными средствами, для оптимизации производительности, в обычных приложениях использование unsafe-режима категорически не рекомендуется. В данном пособии unsafe-режим не рассматривается.

Структуры – это механизм, который является аналогом классов, но реализован на основе стека. Синтаксически структуры и классы в

программе на С# мало отличаются друг от друга. Однако в некоторых случаях использование структур вместо классов позволяет улучшить производительность приложения, иногда очень значительно. Например, массив структур хранится в стеке последовательно, обработка такого фрагмента данных не требует дополнительных переходов. Массив объектов классов хранится в виде массива указателей, при обработке требуется осуществлять дополнительные переходы по указателям, что может существенно увеличить время обработки данных. Поскольку в современном варианте С# структуры в основном являются механизмом оптимизации производительности приложений, то далее в пособии они детально не рассматриваются. Структуры детально описаны в [1, 2].

### **3.2 Базовые типы данных**

Базовые типы данных являются типами-значениями и хранятся в стеке.

Полная система базовых типов детально описана в [1]. В данном разделе мы рассмотрим только наиболее часто используемые типы.

Необходимо отметить, что у каждого типа есть полное наименование, принятое в CTS, а также краткое наименование, используемое в С#. Разработчики могут использовать оба варианта наименования, но по соображениям удобства обычно используют краткие наименования.

Целочисленные типы данных приведены в таблице 1.

Таблица 1. Целочисленные типы данных.

<b>Наименование в С#</b>	<b>Наименование в CTS</b>	<b>Описание</b>
sbyte	System.SByte	Целое 8-битное число со знаком
short	System.Int16	Целое 16-битное число со знаком
int	System.Int32	Целое 32-битное число со знаком
long	System.Int64	Целое 64-битное число со знаком

byte	System.Byte	Целое 8-битное число без знака
ushort	System.UInt16	Целое 16-битное число без знака
uint	System.UInt32	Целое 32-битное число без знака
ulong	System.UInt64	Целое 64-битное число без знака

Типы данных с плавающей точкой приведены в таблице 2.

Таблица 2. Типы данных с плавающей точкой.

<b>Наименование в C#</b>	<b>Наименование в CTS</b>	<b>Описание</b>
float	System.Single	Число с плавающей точкой 32-битное одинарной точности, около 7 значащих цифр после запятой
double	System.Double	Число с плавающей точкой 64-битное двойной точности, около 15 значащих цифр после запятой
decimal	System.Decimal	Число с плавающей точкой 128-битное повышенной точности, около 28 значащих цифр после запятой

Необходимо отметить, что внутреннее представление типа decimal отличается от внутреннего представления типов float и double.

Описание логического типа приведено в таблице 3.

Таблица 3. Логический тип данных.

<b>Наименование в C#</b>	<b>Наименование в CTS</b>	<b>Описание</b>
bool	System.Boolean	Может принимать значения true или false.

Необходимо отметить, что в условных операторах C# и Java, как и в Паскале используется логический тип. В классических C и C++ логический тип отсутствует, его роль выполняет целочисленный тип, хотя в некоторых современных диалектах C++ логический тип используется.

Описание символьного и строкового типов данных приведено в таблице 4.

Таблица 4. Символьный и строковый типы данных.

Наименование в C#	Наименование в CTS	Описание
char	System.Char	Символ Unicode
string	System.String	Строка символов Unicode

Необходимо отметить, что тип string является не типом-значением, а ссылочным типом.

В C# ограничителем для символа является одинарный апостроф, а для строки двойная кавычка. Пример объявления символа и строки:

```
char c = '1';
string str = "строка";
```

Корневым типом в CTS является тип System.Object (в C# – object). От этого типа наследуются все типы, как ссылочные типы, так и типы-значения. Сам тип object является ссылочным.

Таким образом, типы string и object являются исключением из правила, что базовые типы являются типами-значениями.

### 3.3 Преобразования типов

Преобразование типов выполняется аналогично C++ и Java с использованием оператора приведения типов – круглые скобки. Пример:

```
double d1 = 123.45;
int i1 = (int)d1;
```

Очень часто, особенно при консольном вводе, необходимо преобразовать строковое представление числа (введенное с клавиатуры) в числовой формат.

Для этого существует три способа.

Первый способ состоит в том, что у каждого класса-типа существует статический метод `Parse`, преобразующий строку в числовой тип данных.

Пример:

```
int i2 = int.Parse("123");  
double d2 = double.Parse("123,45");
```

Обратите внимание, что метод `Parse` использует настройки локализации операционной системы, в частности в примере типа `double` используется запятая в качестве разделителя целой и дробной части. В том случае если строка не содержит корректное представление числа, метод `Parse` генерирует исключение `FormatException` (исключения подробнее рассматриваются далее в пособии).

Второй способ состоит в том, что вместо метода `Parse` используется метод `TryParse`. В случае неудачного преобразования данный метод не генерирует исключение, но возвращает логическое значение `false`. Возвращаемое число представляется выходным параметром. Выходные параметры (out-параметры) рассматриваются далее в пособии. Пример:

```
int i3;  
bool result = int.TryParse("123", out i3);
```

Третий способ состоит в использовании класса `Convert`. Этот класс содержит большое количество статических методов и может преобразовать значения большинства базовых между собой. Пример:

```
int i4 = Convert.ToInt32("123");
```

В том случае если строка не содержит корректное представление числа, класс `Convert` как и метод `Parse` генерирует исключение `FormatException`.

### 3.4 Использование массивов

Для объявления массивов в C# используется следующий синтаксис:

```
int[] array;
```

В C++ квадратные скобки должны ставить не после имени типа, а после имени переменной «int array[]». В Java допустимы оба варианта объявления.

Если необходимо присвоить начальные значения элементам массива, то форма объявления следующая:

```
int[] array = new int[3] {1, 2, 3};
```

Аналогично для строкового типа:

```
string[] strs =  
    new string[3] { "строка1", "строка2", "строка3" };
```

Многомерные массивы в C# бывают двух видов – прямоугольные и зубчатые (jagged).

Объявление прямоугольных массивов очень похоже на объявление в Паскале. Пример:

```
int[,] matrix = new int[2, 2] { { 1, 2 }, { 3, 4 } };
```

Или:

```
int[,] matrix = { {1,2}, {3,4} };
```

Или:

```
int[,] matrix = new int[2, 2];  
matrix[0, 0] = 1;  
matrix[0, 1] = 2;  
matrix[1, 0] = 3;  
matrix[1, 1] = 4;
```

Объявление зубчатых массивов похоже на объявление в C++ и Java.

Пример:

```
int[][] jagged = new int[3][];  
jagged[0] = new int[3] { 1, 2, 3 };  
jagged[1] = new int[2] { 4, 5 };  
jagged[2] = new int[4] { 6, 7, 8, 9 };
```

В случае зубчатого массива каждый подмассив может иметь собственную размерность.

### **3.5 Консольный ввод/вывод**

Средства консольного ввода/вывода в C# напоминают аналогичные средства в Java и очень уступают по возможностям консольному вводу/выводу в C++.

Ввод и вывод в консоль осуществляется с помощью класса `Console`. У этого класса есть статические методы для ввода и вывода данных:

- `Console.WriteLine` – вывод данных с переводом строки;
- `Console.Write` – вывод данных без перевода строки;
- `Console.Read` – чтение текущего символа;
- `Console.ReadKey` – чтение текущего символа или функциональной клавиши;
- `Console.ReadLine` – чтение строки до нажатия ввода.

У класса `Console` также существуют методы для чтения и изменения цвета текста, размеров окна и другие.

Фактически для ввода данных в консоли существует только метод `Console.ReadLine`, который позволяет ввести только строковый тип данных.

Здесь C# уступает по возможностям консольной библиотеке C++.

Поэтому в C# при разработке консольных приложений актуальным является вопрос по преобразованию строкового типа данных в другие типы.

Методы `Console.WriteLine` и `Console.Write` имеют большое количество перегрузок для вывода данных различных типов.

Также существует перегрузка для форматированного вывода, когда первым параметром является строка формата, а далее передается



произвольное количество параметров, которые подставляются в строку формата. Пример:

```
int p1 = 2;
int p2 = 4;
Console.WriteLine("{0} умножить на {1} = {2}", p1, p2, p1*p2);
```

В консоль будет выведено:

```
2 умножить на 4 = 8
```

Фигурные скобки в строке формата означают ссылку на соответствующий параметр, параметры нумеруются с нуля. Таким образом, вместо {0} будет подставлена переменная p1, вместо {1} переменная p2, вместо {2} выражение p1\*p2.

Еще один пример, содержащий форматирование параметра:

```
double d3 = 1.12345678;
Console.WriteLine("{0} округленное до 3 разрядов = {0:F3}", d3);
```

В консоль будет выведено:

```
1,12345678 округленное до 3 разрядов = 1,123
```

Выражение {0:F3} означает, что нулевой параметр нужно вывести в виде числа с плавающей точкой, округлив до 3 знаков после разделителя разрядов.

### 3.6 Основные конструкции языка программирования C#

Данный раздел опирается на **пример 1**, который приводится целиком.

В дальнейшем в основном будут приводиться только фрагменты примеров.

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

namespace Structures
{
    class Program
    {
        /// <summary>
        /// Использование основных операторов
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
```

```

{
    string str = "строка1";

    //+++++
    // 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);

    //+++++
    // Циклы
    //+++++

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

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

    //Цикл while
    Console.WriteLine("\nЦикл while: ");
    int i3 = 0;

```

```

while (i3 < 3)
{
    Console.Write(i3);
    i3++;
}

//Цикл do while
Console.WriteLine("\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("\n\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 = 0, x3 = 0;
    ParamOut(x, out x2, out x3);
    Console.WriteLine("Вызов функции ParamOut. x={0}, x^2={1}, x^3={2}",
x, x2, x3);

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

    //+++++
    // Использование yield
    //+++++

    Console.Write("\nИспользование yield: ");
    foreach (string st in YieldExample())
        Console.Write(st);

    Console.ReadLine();
}

/// <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);
}

/// <summary>
/// Значения, возвращаемые с помощью yield воспринимаются как значения
массива
/// их можно перебирать с помощью foreach
/// </summary>
/// <returns></returns>
static IEnumerable YieldExample()
{
    yield return "1 ";
    yield return "2 ";
    yield return "333";
}
}
}

```

На основе данного примера рассмотрим основные конструкции языка C#.

### 3.6.1 Пространства имен и сборки

Программа начинается с операторов `using`, каждый оператор указывает пространство имен для библиотечных классов. Каждый класс в языке `C#` должен быть объявлен в каком-либо пространстве имен с использованием оператора `namespace`.

Пространства имен представляют собой древовидную структуру. В каждой ветви данного дерева 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 не подключаются. Поэтому классы вложенных пространств имен необходимо подключать отдельными директивами:

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

Необходимо отметить, что пространства имен представляют собой логическую структуру для систематизации классов. А как эта структура соотносится с откомпилированными классами?

Откомпилированный бинарный код для платформы `.NET` хранится в файлах сборок (`assembly`). Файл сборки может иметь расширение `.dll` или `.exe` по аналогии с библиотеками и исполняемыми файлами ОС `Windows`, но данные файлы содержат бинарный код, который должен выполняться только на платформе `.NET`, если данная платформа не установлена, то ОС `Windows` не сможет запустить такой исполняемый файл.

Файлы сборок необходимо подключать в разделе «References» проекта, что показано на рисунке 3. При нажатии правой кнопкой мыши на пункт «References» открывается диалог по добавлению новых сборок.

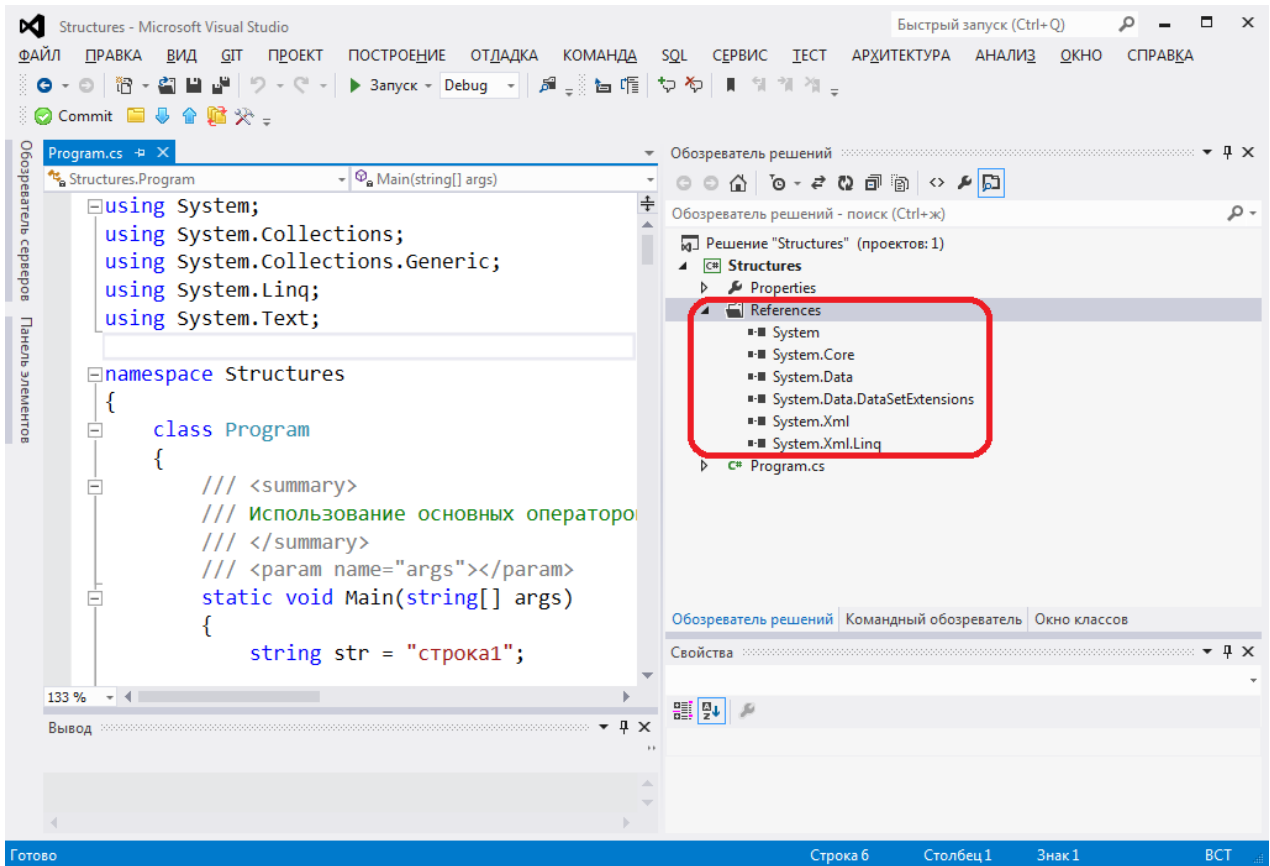


Рис. 3. Файлы сборки.

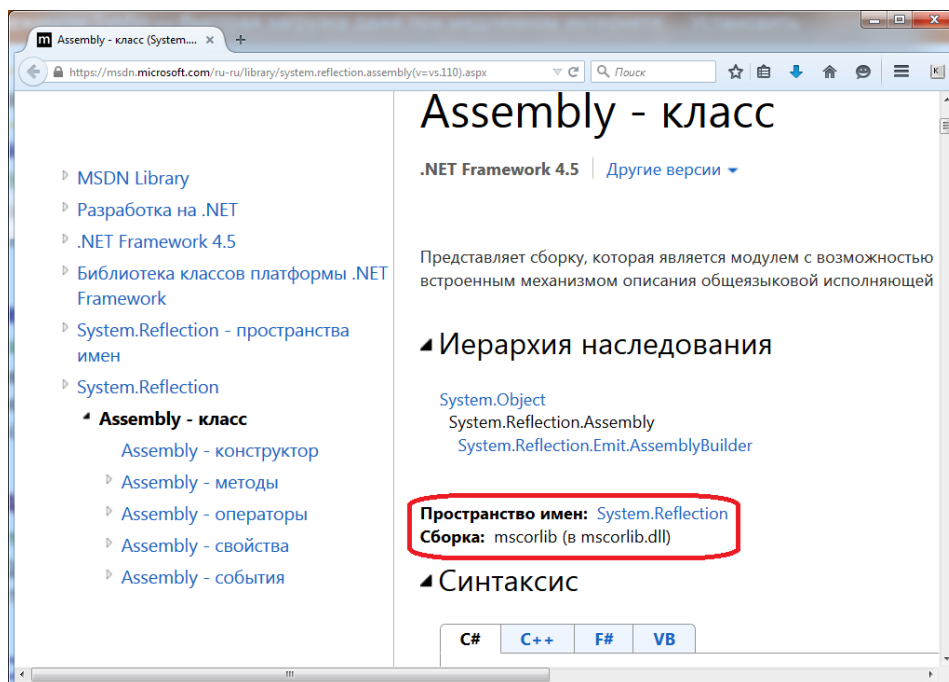


Рис. 4. Указание пространства имен и сборки в справочной системе.

Из рисунка 3 можно сделать вывод о том, что раздел «References» содержит те же значения что и операторы using. Однако это принципиально разная информация. Раздел «References» содержит 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 одного и того же пространства имен содержатся в разных сборках, и наоборот, одна сборка содержит классы из разных пространств имен. Поэтому в справочной системе Microsoft для каждого класса указывается как имя сборки, так и пространство имен, пример на рис 4.

Хотелось бы отметить, что концепция пространства имен является спецификой Microsoft.

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

В Java существует концепция похожая на пространства имен – пакеты (`package`). Каждый класс может быть включен в пакет по аналогии с пространством имен. Но в данной концепции существуют ограничения – пакет является каталогом файловой системы, в который вложены файлы классов, если имя пакета является составным (содержит точку), то это предполагает вложенность соответствующих каталогов. Один файл в Java не может содержать более одного класса, таким образом, файл-класс однозначно соответствует пакету-каталогу. Пространства имен в C# являются более гибким механизмом, однако сторонники Java полагают, что подобная жесткость позволяет избежать ошибок и несоответствий, которые возможны в пространствах имен.

Далее в тексте программы встречаются конструкции:

```
namespace Structures
{
    class Program
    {
        ...
    }
}
```



Команда namespace объявляет пространство имен, а class объявляет класс. Команда namespace может содержать несколько классов. Для того чтобы сослаться на класс Program необходимо использовать директиву:

```
using Structures;
```

Метод:

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

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

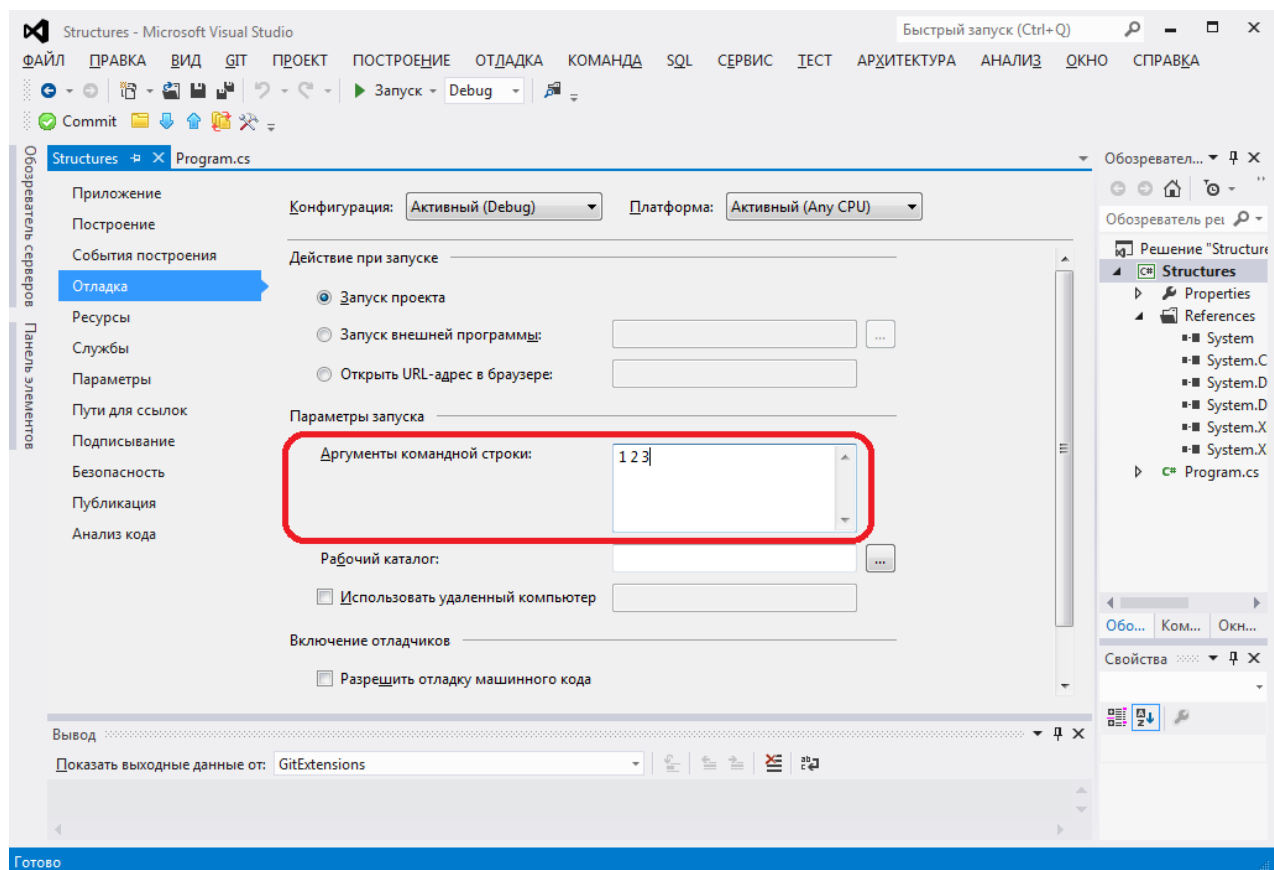


Рис. 5. Задание аргументов командной строки при отладке.

В Visual Studio в целях отладки данные параметры можно указать в свойствах проекта. Для этого необходимо 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 C#. Можно отметить, что они очень похожи на соответствующие конструкции языков C++ и Java.

### 3.6.2 Условные операторы

Условный оператор if:

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

В качестве условия проверяется значение логического типа, такой тип отсутствует в стандартном C++, но присутствует в Паскале, Java, C#.

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

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

Оператор switch выполняется аналогично C++ и Java. Пример:

```
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, после `case` указывается значение проверяемого выражения. 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` завершают оператором `break`. Если ни один оператор `case` не удовлетворяет условию, то выполняются операторы секции `default`.

### 3.6.3 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:

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

В данном примере оператор вывода переменной «`i`» не вложен в фигурные скобки, это не обязательно, так как он является единственным оператором цикла. Если бы в цикле выполнялось несколько действий, то их необходимо было бы вложить в фигурные скобки.

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

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

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

Циклы с предусловием и постусловием также очень похожи на 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);
```

### 3.6.4 Обработка исключений

Если потребовалось бы написать на чистом языке «С» приложение, выполняющее критически важные действия, то оно было бы написано в следующем стиле: производится вызов функции, функция возвращает код возврата (код ошибки), производится анализ кода возврата с помощью оператора switch...case, в зависимости от кода возврата выполняются действия по обработке ошибок. Нет сомнений в том, что такое программирование является очень трудоемким.

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 «С» 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# являются классами,

унаследованными от класса `Exception`. В скобках после `catch` указывают класс исключения и переменную этого класса, из переменной можно получить конкретные сведения о произошедшем исключении.

Блоков `catch` может быть несколько, каждый блок осуществляет перехват исключения определенного вида. В этом случае срабатывает только один блок `catch`, первый по порядку, поэтому порядок обработки исключений имеет значение. Необходимо, чтобы первыми располагались наиболее детальные исключения, те, которые располагаются на наиболее детальном уровне в дереве наследования от класса `Exception`. Если первым в списке расположить 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, файл будет закрыт вне зависимости от того были ошибки при работе с файлом или нет. Пример блока обработки исключений:

```
Console.WriteLine("\n\ndеление 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 в этом случае не происходит. Поэтому разработчик должен искусственно сгенерировать ошибку с помощью команды `throw`. При этом можно генерировать и стандартные исключения, существующие в .NET. Пример:

```

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");
}

```

### 3.6.5 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 имени метода. При объявлении метода указываются формальные параметры, при вызове в них производится подстановка фактических параметров, с которыми осуществляет работу метод при данном вызове.

В C# существует 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. Ключевое слово `ref` является аналогом передачи параметра по указателю в 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, это 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 как при объявлении параметров в методе, так и при вызове метода. Указание этих ключевых слов при вызове метода, очевидно, является излишним для компилятора, компилятору 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 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:

```
/// <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, как и в C++ и Java, используется ключевое слово `return`. Но в C# также существует возможность возвращать из метода несколько значений с использованием конструкции `yield return`. В этом случае метод возвращает последовательность значений, которые могут быть обработаны в вызывающем методе.

Пример вызова метода:

```
Console.WriteLine("\nИспользование yield: ");
foreach (string st in YieldExample())
    Console.WriteLine(st);
```

Пример объявления метода:

```
/// <summary>
/// Значения, возвращаемые с помощью yield воспринимаются как
/// значения массива
/// их можно перебирать с помощью foreach
/// </summary>
static IEnumerable YieldExample()
{
    yield return "1 ";
    yield return "2 ";
    yield return "333";
}
```

Отметим, что конструкцию `yield return` применяют сравнительно редко, так как существует возможность вернуть из метода коллекцию.

### 3.7 XML-комментарии

При объявлении классов, методов и других структур существует возможность 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 слэша, 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 нет необходимости вводить полную структуру XML-комментариев. Для того чтобы добавить XML-комментарий необходимо установить курсор на строку кода перед объявлением функции (класса и т.д.) и три раза набрать прямой слэш «/». После этого автоматически добавляется заголовок XML-комментария.

Пример:

```

/// <summary>
///
/// </summary>
/// <param name="p1"></param>

```

```

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

```

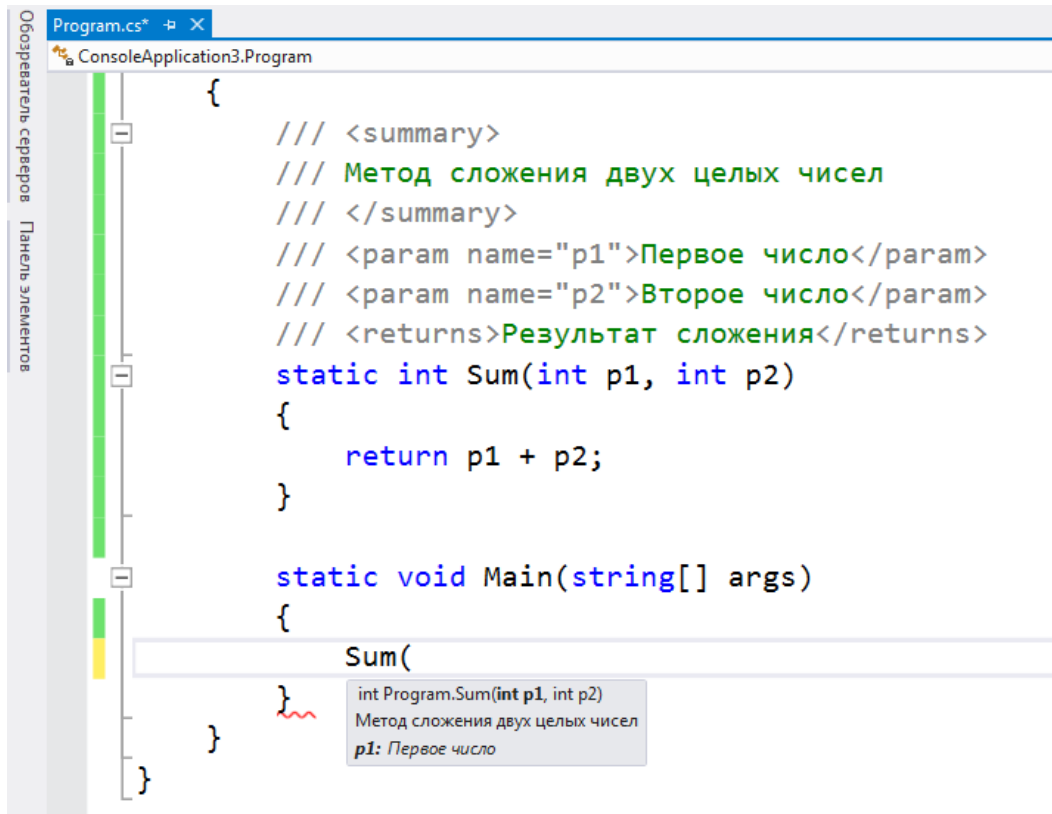


Рис. 6. Использование XML-комментариев при вызове функции Sum.

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

При вызове функции Sum XML-комментарии будут автоматически подсвечены с помощью IntelliSense, рис. 6.

### 3.8 Директивы препроцессора

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

В C# можно использовать директивы `#define` и `#undef` для определения/удаления символа.

Директивы `#if`, `#elif`, `#else`, `#endif` используются для организации условий.

Директивы `#warning` и `#error` организуют выдачи предупреждений и ошибок во время компиляции. Пример:

```
#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:

```
#region Блок кода

    int int1 = 10;

#endregion
```

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

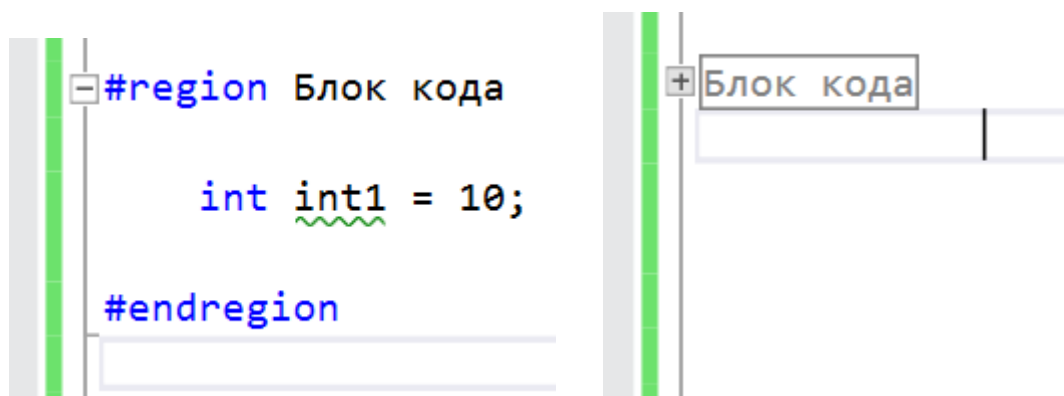


Рис. 7. Результат работы директивы `region` в развернутом и свернутом виде.

### 3.9 Пример консольного ввода/вывода с преобразованием типов данных

В качестве обобщающего примера приведен **пример 2** программы, реализующей консольный ввод/вывод с преобразованием типов данных. В данном примере содержатся управляющие конструкции C# (условные операторы, обработка исключений, вызов методов), преобразование типов данных при консольном вводе/выводе. Пример содержит подробные комментарии.

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

namespace ConsoleInputOutput
{
    class Program
    {
        static void Main(string[] args)
        {
            double d;

            Console.Write("Введите число: ");
            string str = Console.ReadLine();

            //Преобразование строки в число - вариант 1
            Console.WriteLine("Преобразование строки в число 1");
            bool ConvertResult = double.TryParse(str, out d);
            if (ConvertResult)
            {
                Console.WriteLine("Вы ввели число " + d.ToString("F5"));
            }
            else
            {
                Console.WriteLine("Вы ввели не число");
            }

            //Преобразование строки в число - вариант 2
            Console.WriteLine("Преобразование строки в число 2");
            try
            {
                d = double.Parse(str);
                Console.WriteLine("Вы ввели число " + d.ToString("F5"));
            }
            catch (Exception e)
            {
                Console.WriteLine("Вы ввели не число: " + e.Message);
                Console.WriteLine("\nПодробное описание ошибки: " + e.StackTrace);
            }

            //Преобразование строки в число - вариант 3
            Console.WriteLine("Преобразование строки в число 3");
        }
    }
}
```

```

try
{
    d = Convert.ToDouble(str);
    Console.WriteLine("Вы ввели число " + d.ToString("F5"));
}
catch (Exception e)
{
    Console.WriteLine("Вы ввели не число: " + e.Message);
    Console.WriteLine("\nПодробное описание ошибки: " + e.StackTrace);
}

//Вывод параметров командной строки
CommandLineArgs(args);

double a = ReadDouble("Введите коэффициент: ");
Console.WriteLine("Вы ввели коэффициент = " + a);

Console.ReadLine();
}

/// <summary>
/// Вывод параметров командной строки
/// </summary>
static void CommandLineArgs(string[] args)
{
    //Вывод параметров командной строки - вариант 1
    Console.WriteLine("\nВывод параметров командной строки 1:");
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine("Параметр [{0}] = {1}", i, args[i]);
    }

    //Вывод параметров командной строки - вариант 2
    Console.WriteLine("\nВывод параметров командной строки 2:");
    int i2 = 0;
    foreach (string param in args)
    {
        Console.WriteLine("Параметр [{0:F5}] = {1}", i2, param);
        i2++;
    }
}

/// <summary>
/// Ввод вещественного числа с проверкой корректности ввода
/// </summary>
/// <param name="message">Подсказка при вводе</param>
static double ReadDouble(string message)
{
    string resultString;
    double resultDouble;
    bool flag;

    do
    {
        Console.Write(message);
        resultString = Console.ReadLine();

        //Первый способ преобразования строки в число
        flag = double.TryParse(resultString, out resultDouble);
    }
}

```

```

//Второй способ преобразования строки в число
/*
try
{
    resultDouble = double.Parse(resultString);
    //resultDouble = Convert.ToDouble(resultString);
    flag = true;
}
catch
{
    //Необходимо присвоить значение по умолчанию
    resultDouble = 0;
    flag = false;
}
*/

if (!flag)
{
    Console.WriteLine("Необходимо ввести вещественное число");
}
while (!flag);

return resultDouble;
}
}
}

```

В этом примере хотелось бы обратить внимание на функцию `ReadDouble`. Данная функция осуществляет ввод строки с клавиатуры и пытается преобразовать строку в вещественное число. Если такое преобразование невозможно, то строка вводится повторно до тех пор, пока преобразование строки в число не будет успешным.

## 4 Основы ООП в C#

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

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

Разберем основы ООП в C# на **примере 3**.

## 4.1 Объявление класса и его элементов

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

Рассмотрим более детально базовый класс примера – `BaseClass`:

```

/// <summary>
/// Базовый класс
/// </summary>
class BaseClass
{
    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; }
}

```

### 4.1.1 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 переменной класса осуществляется с помощью ключевого слова `this`.

Если конструктор не определен в классе явно, то считается, что у класса есть пустой конструктор без параметров.

#### 4.1.2 Объявление методов

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

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

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

В 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 наследуемых классах текущей сборки.

#### 4.1.3 Объявление свойств

Важным понятием в C# являются свойства. Это понятие отсутствует в явном виде в C++ и Java, вернее реализуется в этих языках с помощью данных и методов.

Если бы в C# не было свойств то в стиле 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 С# для реализации этой задачи существует специальный вид конструкции – свойство. Пример:

```

//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`) переменной, которая 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 это объявление очень похоже на объявление переменной, но после такого объявления ставятся фигурные скобки и указываются секции `get` и `set`. Секции `get` и `set` принято называть аксессорами (`accessors`), поскольку они обеспечивают доступ к свойству.

Get-аксессор (аксессор чтения) является методом, который вызывается при чтении значения свойства. Как правило, он содержит

конструкцию `return`, которая возвращает значение опорной переменной. Конечно, в `get`-аксессоре может выполняться и более сложный код.

Тип возвращаемого значения `get`-аксессора (который в коде явно не задается) 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.

Типом возвращаемого значения `set`-аксессора (который в коде явно не задается) является `void`.

Объявленное свойство ведет себя как переменная, которой можно присвоить значение 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 аксессуара чтения совпадает с областью видимости свойства (`public`), а область видимости аксессуара записи ограничена текущим классом (`private`). То есть прочесть значение такого свойства можно из любого места программы, а присвоить значение только в текущем классе.

Рассмотренный код аксессуаров является очень часто используемым. Для того, чтобы облегчить написание кода в C# для таких «стандартных» свойств, принята упрощенная форма синтаксиса, которая называется 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; }
}
```

Обратите внимание, что в автоопределяемом свойстве опорная переменная (`_property2`) в исходном коде явно не задается, но автоматически генерируется на этапе компиляции и позволяет хранить значение свойства на этапе выполнения программы.

Для аксессуаров автоопределяемого свойства, как и для аксессуаров обычного свойства, можно явно указывать собственные области видимости:

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

Свойство может содержать только один из аксессуаров. Например, возможно 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. Обратите внимание, что для данного свойства также задан XML-комментарий.

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

Свойства позволяют придавать программам на 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 при присвоении переменной нужно производить дополнительный контроль (допустим, целочисленной переменной можно присвоить значения только от 1 до 1000) этот контроль можно поместить в set-аксессор (если условие не выполняется, то генерируется исключение) не внося изменений в вызывающий код.

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

#### 4.1.4 Объявление деструкторов

Классы C# могут содержать деструкторы, которые объявляются аналогично деструкторам в C++. Пример:

```

~BaseClass()
{
    Console.WriteLine("\nДеструктор класса BaseClass");
}

```

}

Однако на практике деструкторы используются редко. Это связано с тем, что основной задачей деструкторов в C++ является очистка памяти. Однако в среде .NET очистка памяти выполняется автоматически с помощью сборщика мусора, поэтому надобность в деструкторах практически отпадает.

#### 4.1.5 Объявление статических элементов класса

Элемент класса в 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, для работы с такими элементами не нужно создавать объект класса.

Вызов статического метода выполняется в формате «Имя\_класса.имя\_метода(параметры)». То есть 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. Аналогичным образом создаются и вызываются статические методы в Java, в C++ вместо точки используется символ удвоенного двоеточия «::».

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

#### 4.2 Наследование класса от класса

В 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 C# аналогичен C++. Для обозначения наследования используется символ двоеточия при объявлении класса, после двоеточия указывается имя базового класса «class ExtendedClass1 : BaseClass».

#### 4.2.1 Вызов конструкторов из конструкторов

При наследовании может возникнуть проблема, связанная с доступом к private полям базового класса. В первую очередь эта проблема может возникнуть 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.

Аналогичным образом может быть вызван другой конструктор текущего класса, в этом случае вместо ключевого слова `base` используется ключевое слово `this`. Пример:

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

В этом случае с помощью 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.

#### 4.2.2 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 `Java` и `C#` существенно различается.

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

Поэтому в `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 далее в пособии.

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

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

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

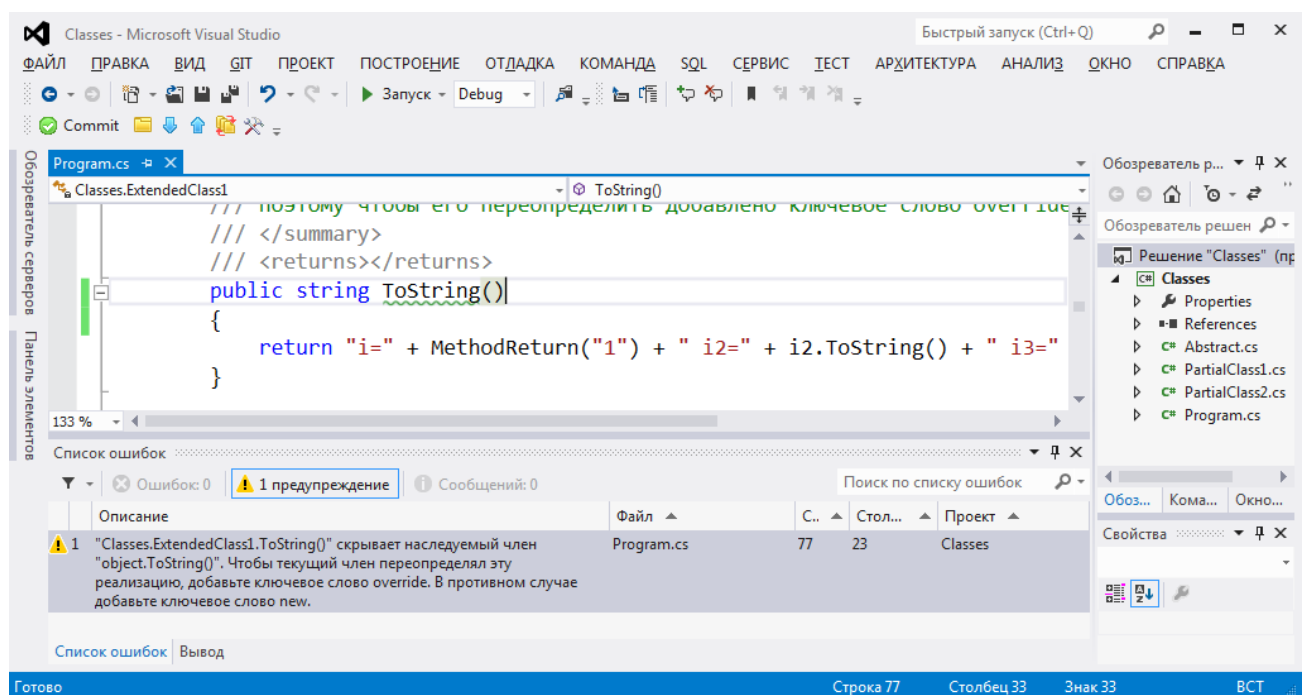


Рис. 8. Предупреждение в случае отсутствия ключевого слова «override».

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 должен использоваться как виртуальный. Если программист хочет показать что данный метод не должен использоваться как виртуальный, но имеет имя, совпадающее с именем виртуального метода, то он должен указать ключевое слово «new».

Принято говорить, что при использовании ключевого слова new происходит «сокрытие» виртуального метода. В случае код будет выглядеть следующим образом:

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

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

### **4.3 Абстрактные классы и методы**

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

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

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

В Java и в C# чистые виртуальные методы называются абстрактными, перед объявлением таких методов указывается ключевое слово `abstract`. Если хотя бы один из методов объявлен абстрактным, то весь класс должен быть объявлен как абстрактный, ключевое слово `abstract` также указывается 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 Rectangle : Figure
{
    public override double Area()
    {
        throw new NotImplementedException();
    }
}
```

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

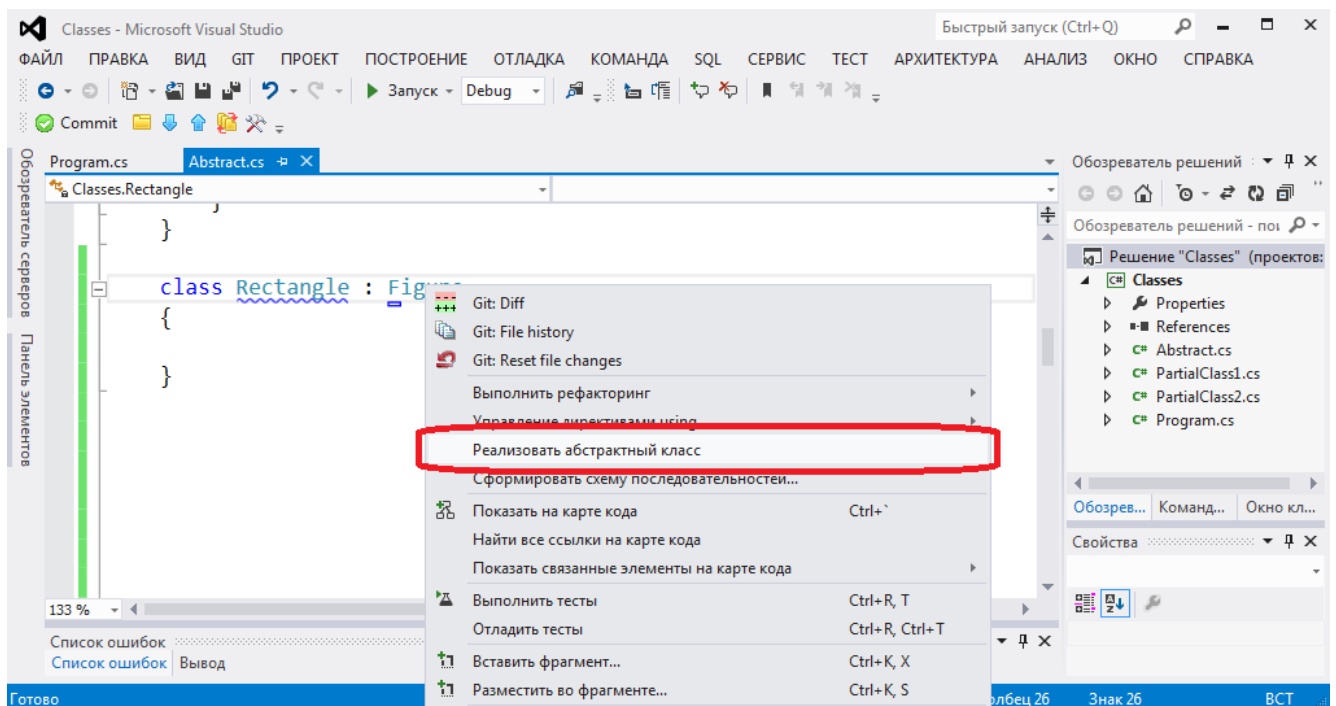


Рис. 9. Реализация абстрактного класса.

#### 4.4 Интерфейсы

Абстрактный класс может содержать как абстрактные методы, так и обычные методы. Обычные методы содержат реализацию в виде кода 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 С# допустимо наследование от нескольких интерфейсов. Почему в С# допустимо наследование от нескольких интерфейсов но только от одного класса?

Наследование от нескольких классов может порождать ряд проблем, наиболее известной из которых является проблема «ромбовидного наследования». На рис 10 приведен пример диаграммы наследования, которая представляет собой ромб. В этом случае классы В и С наследуются от класса А. Класс D наследуется от классов В и С.

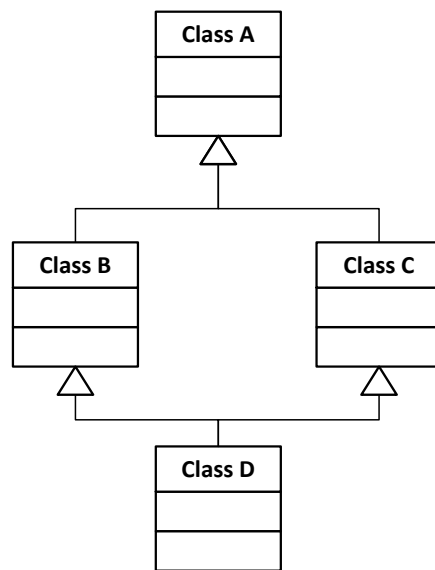


Рис. 10. Пример ромбовидного наследования.

При этом возникает вопрос о том, что делать с полями класса А. Они дублируются при наследовании в классах В и С и могут различным образом изменяться в этих классах. Но класс D может получить доступ к полям класса А, к какой копии полей при этом он получит доступ – к копии класса В или класса С?

Для того чтобы избежать подобных проблем в Java и C# используется наследование только от одного класса. Почему 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, 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, которая наследуется от базового класса и вспомогательная функциональность, которая наследуется от примесей.

При проектировании больших программных систем интерфейсам отводится несколько другая роль. В этом случае они рассматриваются не просто как средство нижнего уровня для устранения ромбовидного наследования, но как высокоуровневое средство, которое позволяет независимо проектировать отдельные подсистемы. Производится декомпозиция системы на подсистемы, 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, который реализует методы.

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

Интерфейсы могут наследоваться от интерфейсов. Пример:

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

В этом случае интерфейс I2 содержит объявление метода I2\_method а также объявление метода I1\_method, который унаследован от интерфейса I1.

#### **4.5 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 C# по аналогии с 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 являются виртуальными, они являются как бы собственными методами 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.

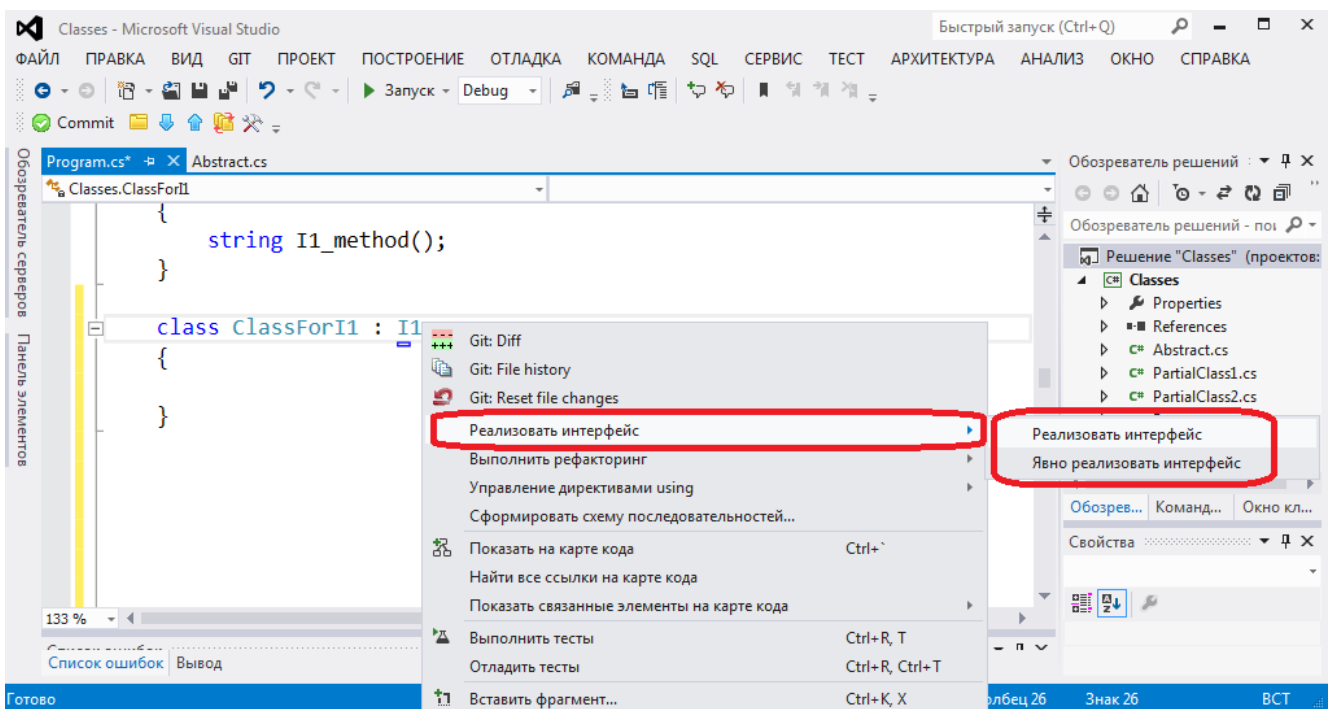


Рис. 11. Реализация интерфейса.

Отличие от абстрактного класса состоит в том, что в случае наследования от интерфейса подменю содержит два пункта «реализовать интерфейс» и «явно реализовать интерфейс». 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();
    }

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

```

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

Сформулируем 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";
    }
}

```

Как используя данную реализацию 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;
```

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

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

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

#### **4.6 Методы расширения**

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

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

Предположим, что мы хотим расширить класс `ExtendedClass2` новым методом, однако по каким-либо причинам нет возможности 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 являться статическим методом. Первым параметром метода расширения является 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.

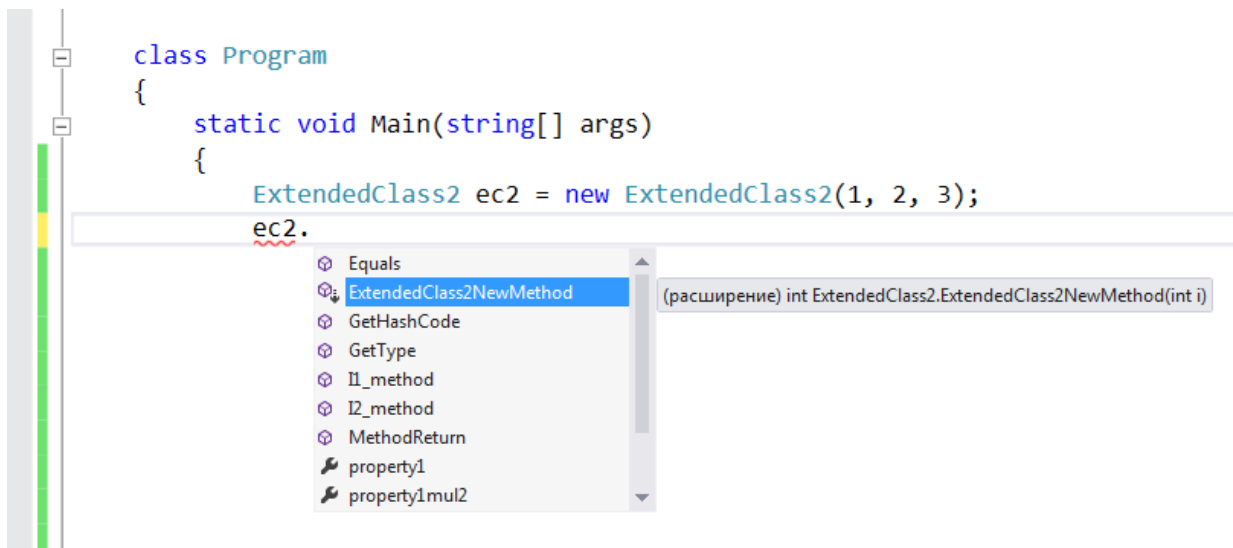


Рис. 12. Работа IntelliSense для метода расширения.

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`, хотя 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.

## 4.7 Частичные классы

Другой уникальный механизм, который существует в C#, это механизм частичных классов. Этот механизм позволяет объявить класс в нескольких файлах.

В C# этот механизм используется вместе со средствами автоматической генерации кода. В Visual Studio существует много средств, которые автоматически генерируют код для обращения к базе данных (технология Entity Framework), для сетевого взаимодействия (технология WCF) и другие.

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

Казалось бы что эта возможность есть в C++, ведь в C++ предусмотрено разделение класса на заголовочный файл и реализацию. Но в C++ невозможно разделить заголовочный файл на несколько файлов. В Java это принципиально невозможно, так как действует правило один класс – один файл. Поэтому когда в Java возникла необходимость создания заглушек для сетевого взаимодействия, то для решения этой проблемы 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.

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.

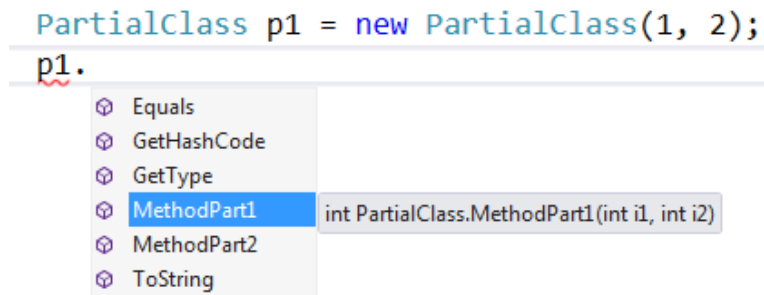


Рис. 13. Работа IntelliSense для частичного класса.

#### 4.8 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.

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

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

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

Нотация диаграмм классов в Visual Studio практически полностью соответствует нотации UML. Наследование классов показано незаштрихованной треугольной стрелкой, реализация интерфейсов стрелкой в виде незаштрихованной окружности.

При нажатии на холсте правой кнопки мыши существует 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.

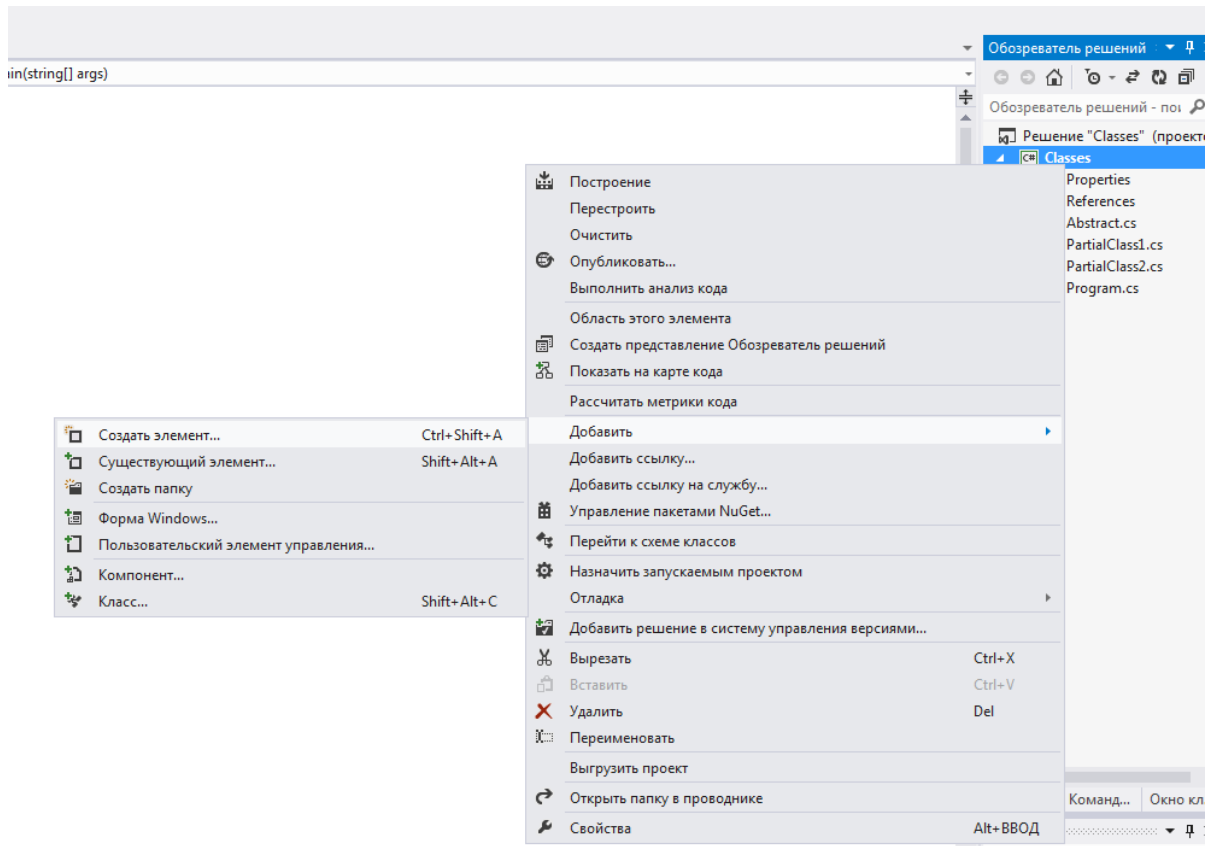


Рис. 14. Добавление диаграммы классов, шаг 1.

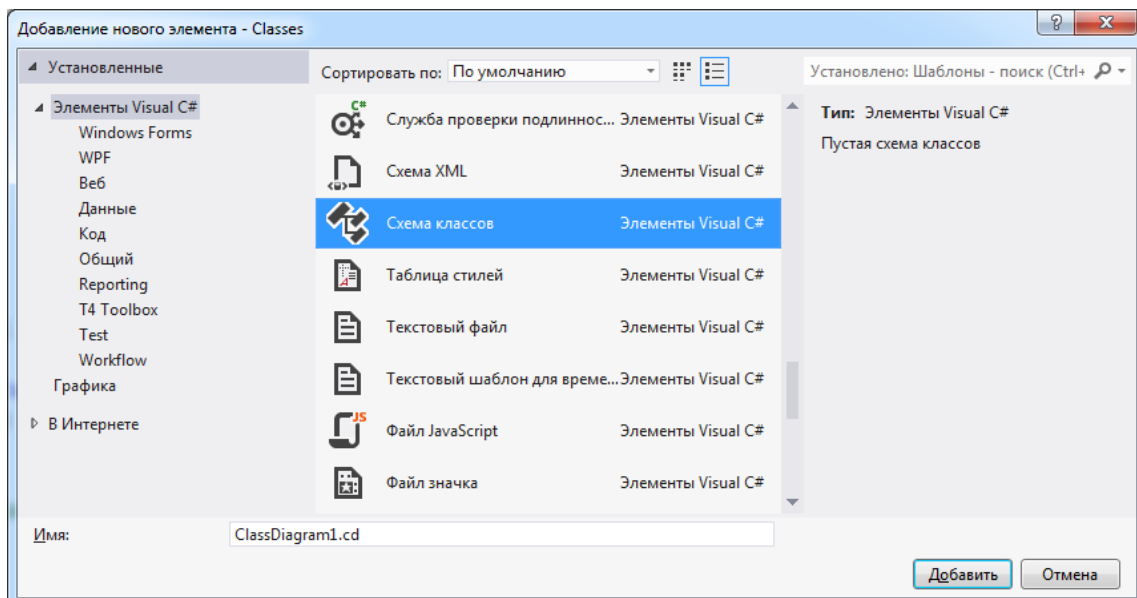


Рис. 15. Добавление диаграммы классов, шаг 2.

В панели кнопок диаграммы классов предусмотрены различные варианты отображения информации о полях классов, что показано на рис. 16. Существуют следующие варианты отображения:

- ТОЛЬКО ИМЕНА полей;
- ИМЕНА И ТИПЫ полей;
- ПОЛНАЯ СИГНАТУРА.

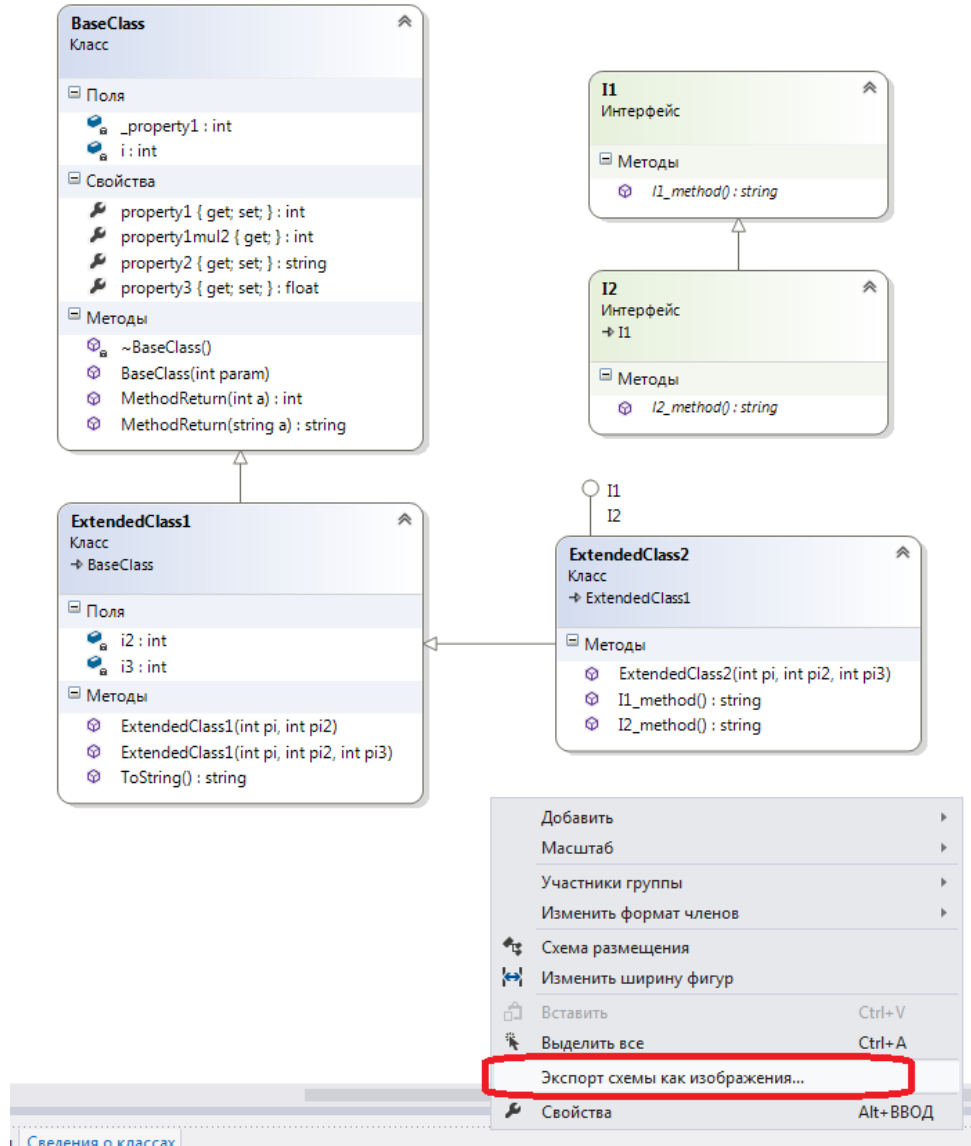


Рис. 16. Пример фрагмента диаграммы классов.

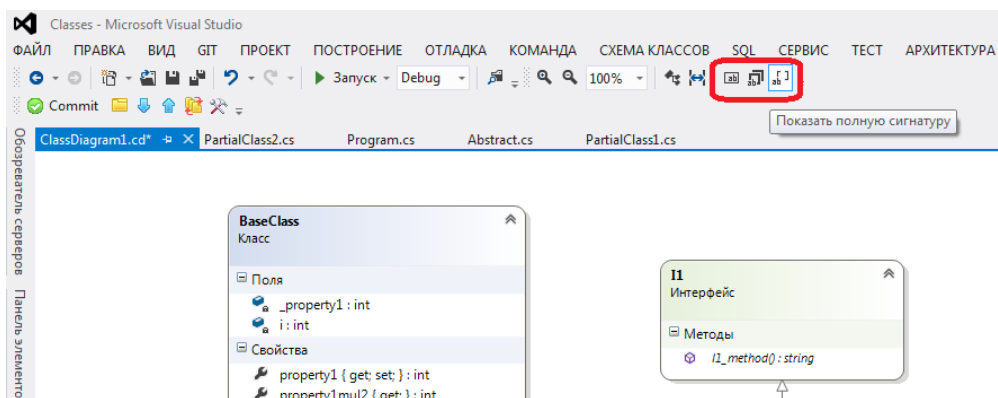


Рис. 17. Кнопки отображения информации о полях классов.



## 4.9 Пример классов для работы с геометрическими фигурами

В качестве обобщающего примера приведен **пример 4** программы, реализующей работу с геометрическими фигурами.

### 4.9.1 Абстрактный класс «Геометрическая фигура»

Основой системы классов для работы с геометрическими фигурами является 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`, которое содержит строковое наименование фигуры. Данное 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. Метод возвращает значение площади (тип `double`) и не принимает параметров, так как вычисление площади должно производиться на основе внутренних данных классов геометрических фигур.

Далее переопределяется виртуальный метод «`ToString`» из класса `object` для приведения к строковому типу.

#### 4.9.2 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().

### 4.9.3 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, то они объявлены в виде обычных полей, а не в виде свойств.

Класс содержит конструктор, который присваивает 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`.

Далее 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».

#### 4.9.4 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 классе фактически реализован только конструктор, который принимает 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.

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

#### 4.9.5 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.

#### 4.9.6 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 консоль будет выведено:

Прямоугольник площадью 20

Квадрат площадью 25

Круг площадью 78,5398163397448

## 5 Расширенные возможности ООП в C#

Помимо базовых возможностей ООП, связанных с созданием классов и интерфейсов и их наследованием, в C# существует большое количество расширенных возможностей, некоторые из них являются уникальными особенностями C#.

### 5.1 Перечисления

Перечисление (enumeration, enum) – это множество целых чисел, при этом за каждым числом закреплена строковая константа.

Перечисление позволяет перечислить множество значений какого-либо свойства.

В классическом C аналогом перечислений были объявления констант с помощью директивы препроцессора #define. Достаточно быстро программисты на C стали объявлять константы с «иерархическими» именами, в имя константы входило и наименование перечисления и наименование конкретного значения. В дальнейшем это привело к появлению перечислений, которые используются в C++, Java и C#.

Рассмотрим работу с перечислениями на **примере 5**.

Объявление перечисления в C#:

```
/// <summary>
/// Перечисление
/// </summary>
public enum СтороныСвета
{
    Север, Юг, Запад, Восток
}
```

При таком объявлении символьным константам Север, Юг, Запад, Восток присваиваются целые значения, начиная с 0.

Поскольку в C# используются символы в формате Unicode, то имена переменных, классов, перечислений и других структур можно задавать на русском языке, как сделано в данном примере.

Присваивать значения символическим константам можно явно:

```
/// <summary>
/// Перечисление с присвоением значений
/// </summary>
public enum СтороныСвета2
{
    Север = 0,
    Юг = 1,
    Запад = 3,
    Восток = 4
}
```

Использование перечислений облегчает понимание кода при проверке условий:

```
СтороныСвета temp1 = СтороныСвета.Восток;
if (temp1 == СтороныСвета.Восток)
{
    Console.WriteLine("Восток");
}
```

Перечисления в C# имеют интересную возможность – значение перечисления можно получить по текстовой строке, которая содержит текст символической константы:

```
СтороныСвета temp2 = (СтороныСвета)Enum.Parse(typeof(СтороныСвета),
"СЕВЕР", true);
```



Метод `Enum.Parse` преобразует строковое значение в значение перечисления. Первым параметром является тип данных перечисления, вторым параметром преобразуемая строка, третьим параметром логическое значение которое показывает нужно ли игнорировать регистр символов строки при преобразовании.

В данном случае регистр символов строки «СЕВЕР» не точно соответствует значению перечисления, но преобразование будет выполнено успешно, так как третий параметр содержит истинное значение. В результате преобразования переменная `temp2` будет содержать значение «СтороныСвета.Север».

## **5.2 Перегрузка операторов**

По возможностям перегрузки операторов `C#` занимает промежуточное значение между `Java` и `C++`.

В `Java` перегрузка операторов не используется, предполагается, что программист на языке `Java` должен использовать методы классов вместо перегруженных операторов.

В `C++` существуют богатые возможности перегрузки операторов. Перегрузка операторов в `C++` разделяется на внутреннюю (когда перегружаемый оператор объявляется как метод экземпляра класса) и внешнюю (когда перегружаемый оператор объявляется как статический метод класса).

В `C#` существует только аналог внешней перегрузки операторов `C++`.

Рассмотрим перегрузку операторов на **примере 6**.

Пример класса, в котором перегружены основные операторы:

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

namespace Operators
{
```

```

/// <summary>
/// Пример класса для перегрузки операторов
/// </summary>
public class Number
{
    /// <summary>
    /// Целое число
    /// </summary>
    public int Num { get; protected set; }

    /// <summary>
    /// Конструктор
    /// </summary>
    public Number(int param)
    {
        this.Num = param;
        Console.WriteLine("Вызов конструктора для {0}", param);
    }

    /// <summary>
    /// Приведение к строке
    /// </summary>
    public override string ToString()
    {
        return this.Num.ToString();
    }

    /// <summary>
    /// Перегрузка унарного оператора
    /// Префиксную и постфиксную формы компилятор различает
автоматически
    /// </summary>
    public static Number operator ++(Number lhs)
    {
        //Не нужно изменять вызываемый объект
        //lhs.Num++;
        //return lhs;

        return new Number(lhs.Num+1);
    }

    /// <summary>
    /// Перегрузка бинарного оператора для (Number + Number)
    /// </summary>
    public static Number operator + (Number lhs, Number rhs)
    {
        int newNum = lhs.Num + rhs.Num;
        Number Result = new Number(newNum);
        return Result;
    }
}

```

```

        //return new Number(lhs.Num + rhs.Num);
    }

    /// <summary>
    /// Перегрузка бинарного оператора для (Number + int)
    /// </summary>
    public static Number operator +(Number lhs, int rhs)
    {
        return new Number(lhs.Num + rhs);
    }

    /// <summary>
    /// Метод как аналог внутренней перегрузки
    /// </summary>
    public void AddInt(int param)
    {
        this.Num += param;
    }

    /// <summary>
    /// Аналог внутренней перегрузки
    /// без создания нового объекта
    /// </summary>
    public static Number operator *(Number lhs, int rhs)
    {
        lhs.Num = lhs.Num * rhs;
        return lhs;
    }

    //Перегрузка операторов сравнения должна производиться попарно
    // == и !=
    // > и <
    // => и <=
    public static bool operator ==(Number lhs, Number rhs)
    {
        return lhs.Num == rhs.Num;
    }

    public static bool operator !=(Number lhs, Number rhs)
    {
        return !(lhs == rhs);
    }

    //Если переопределяется оператор ==, то необходимо переопределить
    //следующие методы
    public override bool Equals(object obj)
    {
        Number param = (Number)obj;
        return this == param;
    }

```

```

public override int GetHashCode()
{
    return this.Num.GetHashCode();
}

//Перегрузка операторов неравенства
public static bool operator < (Number lhs, Number rhs)
{
    return lhs.Num < rhs.Num;
}
public static bool operator > (Number lhs, Number rhs)
{
    return lhs.Num > rhs.Num;
}

public static bool operator <= (Number lhs, Number rhs)
{
    return lhs.Num <= rhs.Num;
}
public static bool operator >= (Number lhs, Number rhs)
{
    return lhs.Num >= rhs.Num;
}

//Операторы приведения типов
//Для одного типа можно задать или явное или неявное приведение
/// <summary>
/// Оператор явного приведения типа Number к типу int
/// </summary>
public static explicit operator int(Number lhs)
{
    return lhs.Num;
}

/// <summary>
/// Оператор неявного приведения типа Number к типу double
/// </summary>
public static implicit operator double(Number lhs)
{
    return (double)lhs.Num;
}

/// <summary>
/// Пример индекатора
/// </summary>
public int this[int i]

```

```

    {
        get
        {
            //Возможный доступ к i-му элементу встроенного массива
            return this.Num + i;
        }
        set
        {
            //Возможный доступ к i-му элементу встроенного массива
            this.Num = value + i;
        }
    }
}

```

Класс содержит свойство Num, которое используется в качестве основы для перегруженных операторов и конструктор, который инициализирует данное свойство. Далее следуют метод приведения к строке ToString и методы, соответствующие перегруженным операторам.

В большинстве перегруженных операторов для обозначения параметров методов используются сокращения: lhs – left hand side (левый оператор), rhs – right hand side (правый оператор).

Как и в C++ для перегрузки операторов используется ключевое слово `operator`. В частности следующее объявление метода:

```
public static Number operator ++(Number lhs)
```

означает, что данный метод соответствует реализации перегруженного оператора ++. Перегрузка операторов в C# может быть реализована только в виде статического метода, метод принимает параметр типа Number и возвращает значение типа Number.

Интересным фактом является то, что компилятор автоматически различает префиксную и постфиксную форму оператора ++. В C++ для различения префиксной и постфиксной перегрузки используется специальный фиктивный параметр типа int. Аналогично реализуется перегрузка оператора --.

В случае перегрузки оператора, который содержит левую и правую части в метод передается два параметра:

```
public static Number operator + (Number lhs, Number rhs)
```

Методы перегрузки операторов поддерживают полиморфизм. Например, можно перегрузить оператор +, который в качестве второго параметра будет принимать не тип Number, а тип int:

```
public static Number operator + (Number lhs, int rhs)
```

Перегрузка операторов в C# предполагает, что возвращаемое значение является новым объектом, например при перегрузке оператора + новый объект создается прямо в операторе return:

```
return new Number(lhs.Num + rhs);
```

Контрпример реализован при перегрузке оператора \*. В этом случае результат сохраняется в левый операнд и левый операнд возвращается из метода. Это является плохой практикой, так как если левый операнд является параметром ссылочного типа, то получается, что оператор неявно меняет один из операндов. Поэтому хорошая практика перегрузки операторов предполагает, что возвращаемое значение является новым объектом, а параметры оператора не изменяются.

Операторы сравнения принимают два операнда, в этом смысле их перегрузка ничем не отличается от рассмотренных примеров. Однако их необходимо перегружать попарно:

- операторы == (равно) и != (не равно);
- операторы > (больше) и < (меньше);
- операторы => (больше или равно) и <= (меньше или равно).

Если один из этих операторов перегружен, а другой нет, то компилятор выдает ошибку, например: *«Для оператора "Operators.Number.operator <(Operators.Number, Operators.Number)" требуется, чтобы был определен соответствующий оператор ">"»*.

Если перегружены операторы равенства и неравенства, то необходимо также переопределить виртуальные методы Equals и GetHashCode. Если методы не переопределены, то компилятор выдает предупреждения:

- *"Operators.Number" определяет оператор == или оператор !=, но не переопределяет Object.GetHashCode().*
- *"Operators.Number" определяет оператор == или оператор !=, но не переопределяет Object.Equals(object o).*

Операторы приведения типов переопределяются несколько необычным способом. Пример оператора приведения к типу int:

```
public static explicit operator int(Number lhs)
```

Здесь вместо оператора указан тип данных int, к этому типу данных будет осуществляться приведение типа.

Перед ключевым словом operator указываются ключевые слова explicit или implicit. Ключевое слово explicit означает, что данный метод должен применяться явно при использовании оператора приведения, implicit означает, что данный метод может применяться неявно при вычислении выражений.

Особенным случаем перегрузки операторов является перегрузка оператора «квадратные скобки». В C# для такой перегрузки используется отдельная конструкция языка, которая называется индексатором. Индексатор напоминает свойство (property), однако у данного свойства предусмотрены параметры, которые указываются в квадратных скобках. Пример:

```
public int this[int i]
{
    get
    {
        return this.Num + i;
    }
    set
    {
        this.Num = value + i;
    }
}
```

В этом случае вместо имени свойства указывается ключевое слово this, после него в квадратных скобках указываются параметры. Параметров может быть произвольное количество и их использование никак не

ограничено. Однако, большинство программистов интуитивно воспринимают квадратные скобки как оператор для доступа к массиву или коллекции, поэтому в качестве параметра обычно передают какой-либо ключ в числовой или символьной форме или набор ключей в случае многомерного массива.

Почему в C# не используется оператор перегрузки квадратных скобок в обычной форме? Потому что форма индекса позволяет перегрузить сразу два оператора чтения и записи, что является наиболее привычным для программиста поведением при перегрузке квадратных скобок.

Рассмотрим пример вызова перегруженных операторов:

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            Number a = new Number(1);

            Console.WriteLine("\nУнарный оператор");

            a++;
            Console.WriteLine("a={0}", a);

            ++a;
            Console.WriteLine("a={0}", a);

            Console.WriteLine("\nБинарный оператор");
            Number b = new Number(100);

            Number c = a + b;
            Console.WriteLine("c={0}", c);

            int i = 333;
            Number d = a + i;
            Console.WriteLine("d={0}", d);

            //Ошибка, так как не перегружен оператор (int + Number)
```



```

//Number d1 = i + d;

Console.WriteLine("\nАналог \"внутренней перегрузки\"
оператора *");
Number mul1 = new Number(5);
Number mul2 = mul1 * 4;
Console.WriteLine("mul1={0}", mul1);
Console.WriteLine("mul2={0}", mul2);

Console.WriteLine("\nПерегрузка операторов сравнения");
Number eq1 = new Number(3);
Number eq2 = new Number(2);

Console.WriteLine("{0} == {1} ----> {2}",
eq1, eq2, (eq1 == eq2));
Console.WriteLine("{0} != {1} ----> {2}",
eq1, eq2, (eq1 != eq2));
Console.WriteLine("{0} > {1} ----> {2}",
eq1, eq2, (eq1 > eq2));
Console.WriteLine("{0} < {1} ----> {2}",
eq1, eq2, (eq1 < eq2));
Console.WriteLine("{0} >= {1} ----> {2}",
eq1, eq2, (eq1 >= eq2));
Console.WriteLine("{0} <= {1} ----> {2}",
eq1, eq2, (eq1 <= eq2));

Console.WriteLine("\nОператоры приведения типов");
Console.WriteLine("Явное приведение типов");
int intEq1 = (int)eq1;
Console.WriteLine("{0} ----> {1}", eq1, intEq1);
Console.WriteLine("Неявное приведение типов");
double doubleEq1 = eq1;
Console.WriteLine("{0} ----> {1}", eq1, doubleEq1);

Console.WriteLine("\nИндексатор");
Number index = new Number(1);
index[330] = 3;
Console.WriteLine("index[{0}] = {1}", 0, index[0]);

Console.ReadLine();
}
}
}

```

Результат вывода в консоль:

Вызов конструктора для 1

Унарный оператор

Вызов конструктора для 2

a=2

Вызов конструктора для 3

a=3

Бинарный оператор

Вызов конструктора для 100

Вызов конструктора для 103

c=103

Вызов конструктора для 336

d=336

Аналог "внутренней перегрузки" оператора \*

Вызов конструктора для 5

mul1=20

mul2=20

Перегрузка операторов сравнения

Вызов конструктора для 3

Вызов конструктора для 2

3 == 2 ---> False

3 != 2 ---> True

3 > 2 ---> True

3 < 2 ---> False

3 >= 2 ---> True

3 <= 2 ---> False

Операторы приведения типов

Явное приведение типов

3 ---> 3

Неявное приведение типов

3 ---> 3

Индексатор

Вызов конструктора для 1  
`index[0] = 333`

Далее в пособии рассматривается пример разреженной матрицы, который использует индексатор.

### 5.3 Обобщения

Обобщения (generics) являются аналогом механизма C++ который называется шаблонами (templates).

Как и шаблоны, обобщения позволяют одинаковым способом выполнять одинаковые действия для различных типов данных.

Однако, реализация обобщений в .NET несколько отличается от реализации шаблонов в компиляторе C++. Если программа на C# содержит обобщенный класс, то он будет скомпилирован в MSIL с обобщенным типом. Подстановка реальных типов вместо обобщенного типа будет производиться уже на этапе JIT-компиляции. Таким образом, работу с обобщениями поддерживает не только компилятор, но и сама среда .NET runtime.

В Java существует аналогичный C# механизм, который также называется обобщениями.

Рассмотрим работу с обобщениями на **примере 7**.

Пример обобщенного класса:

```

/// <summary>
/// Обобщенный класс
/// </summary>
/// <typeparam name="T">Обобщенный тип</typeparam>
class GenericClass1<T>
{
    private T i;

    /// <summary>
    /// Конструктор
    /// </summary>
    public GenericClass1()
    {
        this.i = default(T);
    }
}

```

```

    }

    /// <summary>
    /// Метод инициализации значения
    /// </summary>
    public void SetValue(T param)
    {
        this.i = param;
    }

    /// <summary>
    /// Приведение к строке
    /// </summary>
    public override string ToString()
    {
        return i.ToString();
    }
}

```

Как и шаблоны в C++, обобщенные типы в C# указываются в треугольных скобках после имени класса. Обобщенные типы принято обозначать большими латинскими буквами или начинать с большой буквы.

В данном примере обобщенным типом является тип T.

В классе `GenericClass1` объявлено поле данных «i» обобщенного типа T. Класс содержит конструктор без параметров, в котором осуществляется присваивание начального значения полю «i».

Для присваивания начальных значений переменным обобщенных типов в C# используется специальный оператор `default(тип)`, в качестве параметра передается обобщенный тип. Данный оператор возвращает значение по умолчанию в зависимости от того, какой тип будет подставлен в обобщенный тип, например для числовых типов это «0», для ссылочных типов это «null».

Метод `SetValue` осуществляет присвоение значения полю «i». Метод `ToString` возвращает строковое представление поля «i».

Данный обобщенный класс фактически является контейнером для поля обобщенного типа и приведен в качестве учебного примера. В

дальнейшем в пособии будут рассмотрены примеры коллекций на основе обобщенных типов.

Создадим объекты класса `GenericClass1`:

```
GenericClass1<int> g1 = new GenericClass1<int>();
g1.SetValue(333);
Console.WriteLine("Обобщенный класс 1: " + g1);

GenericClass1<string> g1Str = new GenericClass1<string>();
g1Str.SetValue("строка");
Console.WriteLine("Обобщенный класс 1: " + g1);
```

В данном примере объект `g1` создан в результате подстановки в обобщенный тип типа `int`, а объект `g1Str` создан в результате подстановки в обобщенный тип типа `string`.

В `C#` реализован интересный механизм ограничений, который позволяет накладывать ограничения на обобщенный тип `T`. Список ограничений приведен в таблице 5.

Таблица 5. Список ограничений, поддерживаемых обобщениями.

Ограничение	Описание
<code>where T: struct</code>	Тип <code>T</code> должен быть типом-значением
<code>where T: class</code>	Тип <code>T</code> должен быть ссылочным типом
<code>where T: Class1</code>	Тип <code>T</code> должен наследоваться от класса <code>Class1</code>
<code>where T: Interface1</code>	Тип <code>T</code> должен реализовывать интерфейс <code>Interface1</code>
<code>where T1: T2</code>	При подстановке реальных типов в обобщенные типы <code>T1</code> и <code>T2</code> , тип подставленный в <code>T1</code> должен наследоваться от типа, подставленного в <code>T2</code>
<code>where T: new()</code>	Тип <code>T</code> должен иметь конструктор без параметров

Рассмотрим пример ограничения на основе реализации интерфейса.

Создадим интерфейс:

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

Создадим обобщенный класс, содержащий ограничение:

```
class GenericClass2<T> where T : I1
{
    private T i;

    //Конструктор
    public GenericClass2(T param) { this.i = param; }

    //Приведение к строке
    public override string ToString()
    {
        return i.I1_method();
    }
}
```

Ограничение задается непосредственно при объявлении класса:

```
class GenericClass2<T> where T : I1
```

Для того чтобы создать объект класса `GenericClass2` необходимо сначала создать класс, реализующий интерфейс `I1`.

```
class I1_class : I1
{
    string str;

    public I1_class(string param)
    {
        this.str = param;
    }

    public string I1_method() { return this.str; }
}
```

Теперь возможно создать объект класса `GenericClass2`:

```
GenericClass2<I1_class> g2 =
    new GenericClass2<I1_class>(
        new I1_class("Обобщенный класс с ограничением"));
```

Кроме обобщенных классов в `C#` возможно создавать обобщенные методы. Пример обобщенного метода с ограничением:

```
static void GenericMethod1<T>(T param) where T : I1
{
    Console.WriteLine(param.I1_method());
}
```

```
}
```

Пример вызова обобщенного метода:

```
GenericMethod1<I1_class>(new I1_class("Вызов обобщенного метода"));
```

Обобщенные классы и методы могут содержать несколько обобщенных типов. Пример:

```
static void GenericMethod2<T, Q>(T param1, Q param2)
{
    Console.WriteLine("Обобщенный метод с двумя типами");
}
```

Пример вызова метода:

```
GenericMethod2<I1_class, int>(new I1_class("строка"), 333);
```

Обобщения являются важным и часто используемым механизмом в программах на C#. Кроме обобщенных классов и методов возможно создание других обобщенных структур, например интерфейсов и делегатов.

## **5.4 Делегаты**

Концепция делегата может показаться несколько незнакомой для программиста на C++. Пожалуй наиболее близким аналогом в C и C++ являются указатели на функцию.

Однако эта концепция чрезвычайно понятна программисту на Паскале. Это практически точная копия процедурных типов языка Паскаль.

Преимущество процедурных типов Паскаля и делегатов C# перед указателями на функции C++ состоит в том, что корректность процедурных типов и делегатов проверяется компилятором, который, в частности, может выявить ошибки, связанные с несоответствием параметров. В C++ при работе с указателем на функцию подобные проверки не выполняются, что создает возможность для возникновения ошибок.

В языке Java нет языковой конструкции, подобной делегатам в C#.

Если описывать сущность делегатов кратко, то делегат это аналог типа данных для методов. Класс является типом данных, экземпляром класса является объект, содержащий конкретные данные. Делегат является типом методов, экземпляром делегата является метод, соответствующий делегату.

Рассмотрим работу с делегатами, групповыми делегатами и лямбда-выражениями на **примере 8**.

Пример объявления делегата:

```
delegate int PlusOrMinus(int p1, int p2);
```

Данное описание очень напоминает описание метода (без реализации). В начале описания ставится ключевое слово `delegate`, далее указывается тип возвращаемого значения, далее имя делегата, далее в круглых скобках указываются входные параметры. Имена входных параметров могут быть произвольными, при работе с делегатами они не используются.

Рассмотрим пример методов, соответствующих данному делегату:

```
class Program
{
    static int Plus(int p1, int p2)
    {
        return p1 + p2;
    }

    static int Minus(int p1, int p2)
    {
        return p1 - p2;
    }
}
```

Методы `Plus` и `Minus` соответствуют делегату `PlusOrMinus`, потому что сигнатуры методов совпадают с сигнатурой делегата. То есть данные методы имеют тот же набор параметров и тот же тип возвращаемого значения, которые объявлены в делегате.



Однако, при создании объектов классов с помощью ключевого слова `new` программист явно указывает, экземпляр какого класса он создает, при создании объектов явно указывается имя класса.

При объявлении методов `Plus` и `Minus` не делается никаких ссылок на делегат `PlusOrMinus`. Как же компилятор может определить, что методы соответствуют делегату? Это делается с помощью так называемой неявной типизации.

В этом случае компилятор сравнивает сигнатуры (типы входных параметров и тип возвращаемого значения) метода и делегата. Если сигнатуры совпадают, то считается, что метод соответствует делегату.

В некоторых языках программирования, например в `Ruby`, неявная типизация используется также для классов и объектов. Неявную типизацию иногда называют «утиной» в соответствии с высказыванием, которое называют «утиным тестом»: «если это выглядит как утка, плавает как утка и крикает как утка, то вероятно это утка». То есть если экземпляр имеет те же признаки что и тип, то он подобен этому типу и может считаться экземпляром этого типа.

Следующий вопрос состоит в том, где можно использовать делегаты. Особенность `C#` состоит в том, что делегатный тип может быть использован в качестве параметра метода. Пример:

```
static void PlusOrMinusMethod(
    string str,
    int i1,
    int i2,
    PlusOrMinus PlusOrMinusParam)
{
    int Result = PlusOrMinusParam(i1, i2);
    Console.WriteLine(str + Result.ToString());
}
```

У метода `PlusOrMinusMethod` есть несколько интересных особенностей. Последним параметром данного метода является параметр

делегатного типа. В теле метода имя этого параметра используется как функция, производится ее вызов:

```
int Result = PlusOrMinusParam(i1, i2);
```

При этом метода `PlusOrMinusParam` в программе не существует, будет вызван тот метод, который будет передан в качестве параметра в функцию `PlusOrMinusMethod`.

Таким образом, параметр делегатного типа вызывается как пока неизвестный метод, который в будущем будет передан в качестве параметра.

Сигнатура вызова `PlusOrMinusParam` соответствует сигнатуре делегата: передаются два входных параметра целого типа, возвращается значение целого типа.

Пример вызова метода:

```
int i1 = 3;
int i2 = 2;
PlusOrMinusMethod("Плюс: ", i1, i2, Plus);
PlusOrMinusMethod("Минус: ", i1, i2, Minus);
```

Результат вывода в консоль:

```
Плюс: 5
Минус: 1
```

Обратите внимание, что в данном примере в качестве последнего параметра `PlusOrMinusMethod` передаются имена функций `Plus` и `Minus`.

Рассмотрим способы создания переменных делегатного типа. Пример:

```
PlusOrMinus pm1 = new PlusOrMinus(Plus);
```

Или в сокращенной форме, которую называют «предположением делегата»:

```
PlusOrMinus pm2 = Plus;
```

Создание анонимного метода на основе делегата:

```
PlusOrMinus pm3 = delegate(int param1, int param2)
{
    return param1 + param2;
};
```

Создание анонимных методов применялось до C# версии 3.0, хотя рассмотренный код будет корректно компилироваться и в текущих версиях. Но в C# версии 3.0 для работы с делегатными типами появилось новое средство – лямбда-выражения.

## **5.5 Лямбда-выражения**

Лямбда-выражения или лямбда-функции пришли в C# из функционального программирования. Потребность в них появилась в связи с реализацией технологии LINQ.

Если описывать сущность лямбда-выражений кратко, то это «одноразовые функции». Казалось бы, что такое определение является противоречивым. Ведь в любом классическом учебнике информатики можно прочитать, что методы (процедуры, функции) пишутся один раз и многократно вызываются по имени, что обеспечивает повторное использование кода. Зачем же могут потребоваться одноразовые функции? И почему функцию нельзя объявить традиционным способом и вызывать только один раз, а если потребуется, то и большее количество раз?

Традиционный метод обладает двумя важными свойствами:

- именованностью – имеет имя, по которому метод может быть вызван;
- сигнатурностью – имеет сигнатуру, то есть набор входных параметров с указанием их типов и тип возвращаемого значения.

Лямбда-выражение обладает свойством сигнатурности, но не обладает свойством именованности. То есть у лямбда-выражения есть набор входных параметров и тип возвращаемого значения, но при этом нет имени. Лямбда-выражение записывается в том месте, где оно должно вызываться.

Пример объявления лямбда-выражения:

```
PlusOrMinus pm4 = (int x, int y) =>
{
    int z = x + y;
    return z;
};
```

Лямбда-выражение (подчеркнуто в примере) напоминает по объявлению обыкновенный метод. Объявление начинается со списка входных параметров, которые записываются в скобках. Имя метода в случае лямбда-выражения не указывается. Тип возвращаемого значения определяется компилятором автоматически, используется так называемый механизм вывода типов (англ. type inference).

После скобок указывается стрелка « => », которая отделяет список параметров от тела лямбда-выражения. Стрелка является характерным синтаксическим элементом, по которому всегда можно определить лямбда-выражение. Напомним, что оператор сравнения больше-или-равно записывается как « >= » и не совпадает с лямбда-выражением.

После стрелки записывается тело метода, которое в данном примере ничем не отличается от тела обычного метода.

После такого объявления лямбда-выражение может быть вызвано как метод через определенную переменную pm4:

```
int test = pm4(1, 2);
```

Однако чаще всего лямбда-выражение передают в качестве параметра делегатного типа:

```
PlusOrMinusMethod(
    "Создание экземпляра делегата на основе лямбда-выражения 1: ",
    i1,
    i2,
    (int x, int y) =>
    {
        int z = x + y;
        return z;
    }
);
```

В данном примере лямбда-выражение (подчеркнуто) передается в качестве параметра делегатного типа PlusOrMinus.

Обратите внимание, что соответствие сигнатуры делегата и сигнатуры лямбда-выражения проверяется с помощью неявной типизации.

Например, зададим в лямбда-выражении третий входной параметр, что нарушит соответствие между сигнатурой лямбда-выражения и сигнатурой делегата (третий параметр подчеркнут):

```
PlusOrMinusMethod(
    "Создание экземпляра делегата на основе лямбда-выражения 1: ",
    i1,
    i2,
    (int x, int y, int k) =>
    {
        int z = x + y ;
        return z;
    }
);
```

При этом компилятор выдаст следующую ошибку в строке определения лямбда-выражения: *«Делегат "Delegates.PlusOrMinus" не принимает "3" аргументов»*.

Язык C# предоставляет возможность упрощения синтаксиса лямбда-выражений. Например, вместо исходного лямбда-выражения:

```
(int x, int y) =>
{
    int z = x + y;
    return z;
}
```

можно записать следующее:

```
(x, y) =>
{
    return x + y;
}
```

В данном случае у входных параметров не указываются типы, компилятор определит их автоматически с помощью механизма вывода типов. Возвращаемое значение вычисляется в операторе return. В том

случае, когда результат лямбда-функции может быть задан в виде единого выражения, синтаксис можно еще более упростить:

```
(x, y) => x + y
```

В данном случае компилятор автоматически определяет, что выражение `x + y` является возвращаемым значением лямбда-функции.

Еще одной интересной особенностью лямбда-выражений является возможность использования в них внешних переменных. Пример:

```
int outer = 100;
PlusOrMinus pm5 = (int x, int y) =>
{
    int z = x + y + outer;
    return z;
};
```

Переменная `outer`, которая объявлена на уровне внешней функции, видна в лямбда-выражении и может быть использована в вычислениях.

Таким образом, лямбда-выражение можно рассматривать как фрагмент кода, которым прикладной программист может дополнить функциональность какого-либо библиотечного метода. При этом данный код пишется не произвольно, а имеет интерфейс с библиотечным методом в виде соответствующего делегатного типа.

## 5.6 Обобщенные делегаты *Func* и *Action*

Появление лямбда-выражения в языке `C#` облегчило создание параметров делегатного типа. Однако сами делегаты при этом по-прежнему необходимо создавать отдельно. Нельзя ли облегчить решение и этой задачи? Это сделано в `C#` версии 3.5 с использованием обобщенных делегатов `Func` и `Action`.

Ранее был объявлен следующий делегат:

```
delegate int PlusOrMinus(int p1, int p2);
```

Ему соответствует следующее объявление делегата `Func`:

```
Func<int, int, int>
```

Рассмотренный ранее метод:

```
static void PlusOrMinusMethod(
    string str,
    int i1,
    int i2,
    PlusOrMinus Param)
{
    int Result = PlusOrMinusParam(i1, i2);
    Console.WriteLine(str + Result.ToString());
}
```

может быть переписан следующим образом с использованием обобщенного делегата Func:

```
static void PlusOrMinusMethodFunc(
    string str,
    int i1,
    int i2,
    Func<int, int, int> PlusOrMinusParam)
{
    int Result = PlusOrMinusParam(i1, i2);
    Console.WriteLine(str + Result.ToString());
}
```

Вызов делегатного параметра в теле обоих методов производится совершенно одинаково: параметр вызывается как функция, которой передается два целочисленных параметра и возвращается целочисленный параметр.

В качестве значения параметра, который соответствует обобщенному делегату, может быть использовано имя обычного метода (в примере используется ранее рассмотренный метод Plus):

```
PlusOrMinusMethodFunc(
    "Создание экземпляра делегата на основе метода: ",
    i1,
    i2,
    Plus
);
```

Однако чаще для этих целей используется лямбда-выражение (подчеркнуто в примере):

```

PlusOrMinusMethodFunc(
    "Создание экземпляра делегата на основе лямбда-выражения 3:",
    i1,
    i2,
    (x, y) => x + y
);

```

Таким образом, использование обычного делегата и обобщенного делегата Func позволяет получить одинаковые результаты. Однако преимущество использования Func состоит в том, что его, в отличие от обычного делегата, не нужно объявлять в тексте программы, он уже объявлен в библиотечных классах C#.

Если нажать правую кнопку мыши на ключевом слове Func и выбрать в контекстном меню пункт «перейти к определению», то Visual Studio осуществит переход к следующему определению, которое объявлено в пространстве имен System:

```
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

В языке C# делегаты могут быть обобщенными, и Func является именем обобщенного делегата. При объявлении обобщенного делегата обобщенные типы в треугольных скобках задаются после имени делегата. В данном случае используется три обобщенных типа: два первых типа – это типы входных параметров, а последний – тип возвращаемого значения.

Таким образом, в объявлении «Func<int, int, int>» последний параметр задает тип возвращаемого значения, предыдущие параметры задают входные типы. Это является общим правилом при использовании обобщенного делегата Func – последний параметр определяет тип выходного значения.

Если перейти к определению аналогичного делегата Action:

```
Action<int, int, int>
```

то оно будет следующим:

```
public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
```



В отличие от делегата Func, у которого последний обобщенный параметр задает тип выходного значения, все обобщенные параметры делегата Action задают типы входных параметров. Обобщенный делегат Action возвращает значение типа void.

По сравнению с разбиравшимися ранее примерами обобщенных классов и методов у данных обобщенных делегатов есть еще одна особенность – перед обобщенными типами указаны ключевые слова in и out.

Данная особенность обобщенных делегатов появилась в C# 4.0 и называется ковариантностью и контрвариантностью. Кроме обобщенных делегатов данная особенность может применяться в обобщенных интерфейсах.

Обобщенный делегат или интерфейс ковариантен, если обобщенный тип аннотирован ключевым словом out. Обобщенный тип T является ковариантным, если для подставленного в него конкретного типа t, в параметр типа T может быть передано значение типа t или любого типа, производного от t, то есть данные более конкретного типа, чем изначально заданный. Использование ключевого слова out также означает, что тип T разрешен только в качестве типа возвращаемого значения.

Обобщенный делегат или интерфейс контрвариантен, если обобщенный тип аннотирован ключевым словом in. Обобщенный тип T является контрвариантным, если для подставленного в него конкретного типа t, в параметр типа T может быть передано значение типа t или любого типа, базового для t, то есть данные более общего типа, чем изначально заданный. Использование ключевого слова in также означает, что тип T разрешен только в качестве типа входного параметра.

Если ключевые слова in или out не заданы, то обобщенный тип T является инвариантным, то есть для подставленного в него конкретного

типа  $t$ , в параметр типа  $T$  может быть передано значение только типа  $t$  и никакого другого типа.

Использование ковариантных и контрвариантных делегатов и интерфейсов детально рассмотрено в [1, 2].

### 5.7 Групповые делегаты

Если делегат имеет тип возвращаемого значения `void`, то на его основе может быть создан групповой делегат, которому соответствует сразу несколько методов. В частности этому условию удовлетворяет обобщенный делегат `Action`.

Рассмотрим следующий пример. Определим методы в виде лямбда-выражений, которые соответствуют делегату «`Action<int, int>`», то есть принимают два входных параметра типа `int` и тип возвращаемого значения `void`:

```
Action<int, int> a1 =
    (x, y) =>
        { Console.WriteLine("{0} + {1} = {2}", x, y, x + y); };

Action<int, int> a2 =
    (x, y) =>
        { Console.WriteLine("{0} - {1} = {2}", x, y, x - y); };
```

Теперь на основе данных методов можно определить групповой делегат:

```
Action<int, int> group = a1 + a2;
```

Если вызвать данный групповой делегат:

```
group(5, 3);
```

то будут вызваны методы `a1` и `a2`. В консоль будет выведено следующее:

```
5 + 3 = 8
```

```
5 - 3 = 2
```

Необходимо учитывать, что очередность вызовов метода в групповом делегате не гарантируется. То есть разработчик может быть уверен, что

оба метода `a1` и `a2` будут вызваны, но они могут быть вызваны в произвольном порядке.

Для добавления вызова метода к групповому делегату может быть использован оператор «+=», а для удаления вызова метода из группового делегата оператор «-=». Пример:

```
Action<int, int> group2 = a1;
Console.WriteLine("Добавление вызова метода к групповому делегату");
group2 += a2;
group2(10, 5);

Console.WriteLine("Удаление вызова метода из группового делегата");
group2 -= a1;
group2(20, 10);
```

Результат вывода в консоль:

Добавление вызова метода к групповому делегату

10 + 5 = 15

10 - 5 = 5

Удаление вызова метода из группового делегата

20 - 10 = 10

Групповые делегаты являются основой для реализации механизма событий.

## 5.8 События

События в C# реализуют механизм подписки/публикации. Любой класс может объявить события и вызывать их в необходимые моменты времени. Внешние классы могут прикрепить методы в качестве обработчиков событий или, как говорят «подписаться на событие».

События основаны на обобщенных делегатах, соответственно обработчики событий должны соответствовать сигнатуре обобщенного делегата для события.

В C++ и Java существует большое количество библиотек, реализующих подход подписки/публикации, однако они реализованы в

виде внешних библиотек, а не в виде конструкции, встроенной в язык программирования.

Особенностью событий C# является то, что они соответствуют механизму событий .NET-фреймворка. Например, событие нажатия на кнопку в технологии Windows Forms является событием C#, создание обработчика события нажатия на кнопку происходит с использованием стандартного механизма событий в C#.

Рассмотрим работу с событиями на **примере 9**.

Объявим делегат, на котором основано событие:

```
public delegate void NewEventDelegate(string str);
```

Делегат принимает один строковый параметр. Делегат, на котором основано событие, должен возвращать значение void.

Событие объявлено в классе консольного приложения Program:

```
public static event NewEventDelegate NewEvent;
```

Событие объявляется как обычный элемент класса, фактически NewEvent является специализированной переменной.

Для события указана область видимости (public). Объявление static означает, что данное поле является статическим, события также могут быть объявлены в виде нестатических полей. Далее указывается ключевое слово event, затем наименование делегата, на котором основано событие, в данном примере NewEventDelegate. После наименования делегата указывается имя события NewEvent, фактически это имя переменной.

События отображаются в IntelliSense в виде пиктограммы с «молнией», что показано на рис. 18.

Для прикрепления обработчика к событию используется перегруженный оператор «+=», для открепления обработчика от события (удаления обработчика события) перегруженный оператор «-=»:

Поскольку механизм событий основан на механизме групповых делегатов, то к событию может быть прикреплено несколько

обработчиков. Гарантируется, что при вызове события будут вызваны все обработчики, но порядок их вызова не гарантируется. Таким образом, при написании обработчиков событий, программист не должен рассчитывать, что обработчики событий обязательно будут вызваны в порядке их прикрепления к событию.

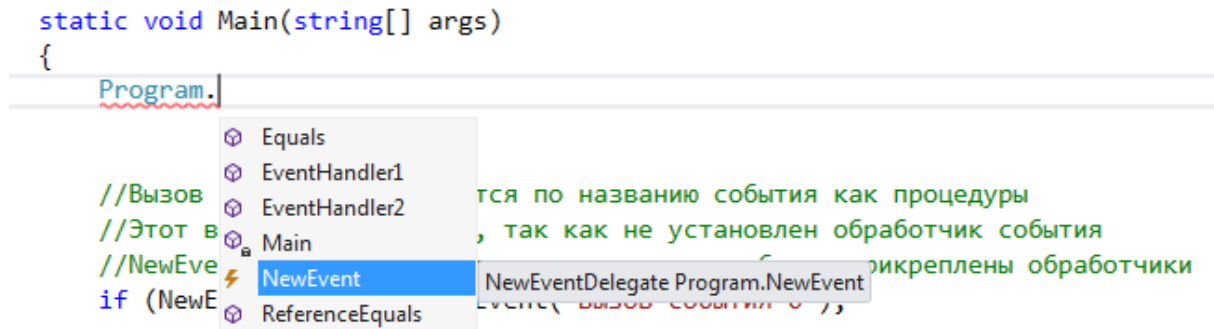


Рис. 18. Отображение события в IntelliSense.

Вызов события производится как вызов обыкновенного метода, в скобках указываются параметры вызова. Однако при вызове событий существует особенность. Если вызвать событие, к которому не прикреплен ни один обработчик, то возникнет исключение «NullReferenceException». Для решения этой проблемы у событий переопределено сравнение с null. Если оператор сравнения события с null возвращает истину, значит к нему не прикреплен ни один обработчик. Поэтому перед вызовом события необходимо производить проверку на сравнение с null.

Объявление обработчиков событий:

```
public static void EventHandler1(string str)
{
    Console.WriteLine("Первый обработчик события: " + str);
}

public static void EventHandler2(string str)
{
    Console.WriteLine("Второй обработчик события: " + str);
}
```

Пример прикрепления обработчиков и вызова события:

```
//Этот вызов не работает, так как не прикреплены обработчики события
//NewEvent != null - проверка того, что к событию прикреплены обработчики
if (NewEvent != null) NewEvent("вызов события 0");
```

```

//Прикрепление обработчика события
Program.NewEvent += EventHandler1;
if (Program.NewEvent != null) Program.NewEvent("вызов события 1");

//Прикрепление второго обработчика события
//Вызываются оба обработчика
Program.NewEvent += EventHandler2;
if (Program.NewEvent != null) Program.NewEvent("вызов события 2");

//Удаление второго обработчика события
Program.NewEvent -= EventHandler2;
if (Program.NewEvent != null) Program.NewEvent("вызов события 3");

```

Результаты вывода в консоль:

```

Первый обработчик события: вызов события 1
Первый обработчик события: вызов события 2
Второй обработчик события: вызов события 2
Первый обработчик события: вызов события 3

```

Рассмотренный пример содержит основы работы с событиями и необходим для понимания основ работы с событиями, однако, хотелось бы отметить следующее:

- как правило, событие объявляется внутри класса и не является статическим, а обработчики события прикрепляются снаружи класса;
- событие основывают на библиотечном делегате EventHandler или других специализированных делегатах;
- обработчики событий часто задают в виде лямбда-выражений.

Рассмотрим пример создания события с учетом данных особенностей. Данный пример основан на стандартном делегате EventHandler, который объявлен в пространстве имен System:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Данный делегат является обобщенным. Он возвращает тип void, что является необходимым условием для делегата события, и принимает два параметра. Первый параметр «sender» типа «object» содержит ссылку на объект, который инициировал событие (напомним, что object является

ссылочным типом). Второй параметр «е» обобщенного типа «EventArgs» содержит произвольные параметры события. В качестве обобщенного типа EventArgs может использоваться любой тип, содержащий описание параметров события.

Рассмотрим класс, предназначенный для описания параметров события:

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

namespace Events
{
    /// <summary>
    /// Параметры события
    /// </summary>
    public class NewGenericEventArgs : EventArgs
    {
        /// <summary>
        /// Конструктор
        /// </summary>
        /// <param name="param"></param>
        public NewGenericEventArgs(string param)
        {
            this.NewGenericEventArgsParam = param;
        }

        /// <summary>
        /// Свойство, содержащее параметр
        /// </summary>
        public string NewGenericEventArgsParam { get; private set; }
    }
}
```

Класс NewGenericEventArgs является наследником класса EventArgs. Библиотечный класс EventArgs является базовым для классов, содержащих данные о событии, и объявлен в пространстве имен System.Runtime.InteropServices.

Класс NewGenericEventArgs содержит автоопределяемое свойство NewGenericEventArgsParam типа string, предназначенное для хранения

строкового параметра. Также класс содержит конструктор, который инициализирует данное свойство.

Рассмотрим класс издателя события `NewGenericEventPublisher`, содержащего событие:

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

namespace Events
{
    /// <summary>
    /// Класс, содержащий событие
    /// </summary>
    public class NewGenericEventPublisher
    {
        /// <summary>
        /// Событие создается через обобщенный делегат
        /// </summary>
        public event EventHandler<NewGenericEventArgs> NewGenericEvent;

        /// <summary>
        /// Вызов события
        /// </summary>
        public void RaiseNewGenericEvent(string param)
        {
            //Если у события есть подписчики
            if (NewGenericEvent != null)
            {
                //Вызов события
                NewGenericEvent(this, new NewGenericEventArgs(param));
            }
        }
    }
}
```

Для объявления события используется обобщенный делегат `EventHandler`. Класс `NewGenericEventArgs` используется в качестве класса-обобщения для хранения параметров события.

```
public event EventHandler<NewGenericEventArgs> NewGenericEvent;
```

Класс `NewGenericEventPublisher` не содержит конструктор, поэтому используется конструктор по умолчанию, не содержащий аргументов.

Для тестирования вызова события используется метод `RaiseNewGenericEvent`. Метод принимает строковый аргумент, который используется при создании параметра события, экземпляра класса



`NewGenericEventArgs`. В методе проверяется, что для события заданы обработчики «`NewGenericEvent != null`», если обработчики заданы, то производится вызов события.

Данный класс является учебным примером, в нем вызов события осуществляется с помощью тестового метода, в реальном классе вызовы событий могут производиться внутри методов класса при достижении определенных условий.

Классом слушателя события будет основной класс `Program` консольного приложения. Работа с событием происходит в основном методе `Main` консольного приложения.

Для вызова события необходимо сначала создать объект класса:

```
NewGenericEventPublisher ne = new NewGenericEventPublisher();
```

Далее к событию, которое является полем объекта класса, прикрепляется обработчик, который в настоящем примере задается в виде лямбда-выражения:

```
ne.NewGenericEvent += new EventHandler<NewGenericEventArgs>(
    (sender, e) =>
    {
        Console.WriteLine(e.NewGenericEventArgsParam);
    }
);
```

Лямбда-выражение соответствует делегату `EventHandler`. В лямбда-выражении осуществляется вывод в консоль строки параметров, передаваемых через объект класса `NewGenericEventArgs`.

Для вызова события в данном примере используется метод `RaiseNewGenericEvent`:

```
ne.RaiseNewGenericEvent("Событие NewGenericEventPublisher");
```

Отметим, что в данном упрощенном примере вызов события происходит в результате внешнего вызова метода `RaiseNewGenericEvent`. Однако, это не совсем естественный путь использования событий. Предполагается, что событие будет вызываться внутри класса при

срабатывании определенных условий, которые характеризуют, например, изменение состояния класса.

В завершении рассмотрения событий отметим, что несмотря на то, что данный механизм активно используется в C#, он может вызывать проблему при сборке мусора. Проблема возникает за счет того, что издатель и слушатель события соединяются напрямую. Для решения этой проблемы используется более сложный шаблон так называемых «слабых событий» (weak events), который использует более гибкий механизм соединения между издателем и слушателем события и не создает проблем при сборке мусора. Этот механизм в частности используется для обработчиков событий в Windows Presentation Foundation.

## **6 Работа с коллекциями**

Коллекции являются важной частью любого языка программирования. В C# существует развитая система коллекций. В данном разделе сначала рассматриваются примеры использования стандартных коллекций, далее рассматриваются примеры создания нестандартных коллекций.

### **6.1 Стандартные коллекции**

Подходы к работе с коллекциями в C++, Java и C# в целом совпадают. Все коллекции можно разделить на две большие категории – обобщенные (шаблонизированные в C++) и необобщенные.

Использование обобщенной коллекции предполагает, что в коллекцию помещаются элементы обобщенного типа, соответственно все элементы коллекции должны быть одного и того же типа, который указывается при создании обобщенной коллекции. В соответствии с принципами ООП вместо указанного обобщенного типа могут

использоваться производные типы. Классы обобщенных коллекций находятся в пространстве имен `System.Collections.Generic`.

Использование необобщенной коллекции предполагает, что в коллекцию помещаются элементы типа `object` (самого базового типа в иерархии типов). Поэтому в коллекцию можно помещать элементы различных типов. Основные классы необобщенных коллекций находятся в пространстве имен `System.Collections`.

Список основных классов коллекций приведен в таблице 6.

Таблица 6. Список основных классов коллекций.

<b>Коллекция</b>	<b>Обобщенный класс</b>	<b>Необобщенный класс</b>
Список	<code>List&lt;T&gt;</code>	<code>ArrayList</code>
Стек	<code>Stack&lt;T&gt;</code>	<code>Stack</code>
Очередь	<code>Queue&lt;T&gt;</code>	<code>Queue</code>
Множество	<code>HashSet&lt;T&gt;</code>	–
Ассоциативный массив (хранящий пары ключ-значение). Такую структуру называют также словарем или хэш-таблицей	<code>Dictionary&lt;TKey, TValue&gt;</code>	<code>Hashtable</code>

Следует отметить, что обобщенные коллекции на практике используются чаще. Поэтому в шаблоне консольного приложения в `Visual Studio` по умолчанию подключено пространство имен обобщенных коллекций `System.Collections.Generic`, а пространство имен необобщенных коллекций `System.Collections` не подключено, в случае необходимости его требуется добавить вручную с помощью ключевого слова `using`.

В таблице 6 приведены только основные коллекции. В .NET существуют десятки классов коллекций, в том числе различные специализированные коллекции для решения конкретных задач (например, битовые коллекции), коллекции для безопасного многопоточного доступа и т.д. В данном пособии они не рассматриваются. Информация о них детально рассмотрена в [1, 2].

Рассмотрим работу со стандартными коллекциями на **примере 10**.

### 6.1.1 Обобщенный список

Рассмотрим пример работы с обобщенным списком. Создание объекта списка:

```
List<int> li = new List<int>();
```

Добавление элементов в список:

```
li.Add(1);
li.Add(2);
li.Add(3);
```

Вывод элементов списка:

```
foreach (int i in li)
{
    Console.WriteLine(i);
}
```

В данном примере в качестве класса-обобщения используется `int`, поэтому создается список целых чисел. Преимущества использования обобщенной коллекции состоят в том, что тип элемента известен как при добавлении, так и при выводе. При добавлении Visual Studio подсказывает тип добавляемого элемента с помощью механизма IntelliSense. Тип элемента уже известен, так как он был ранее указан при создании списка.

При выводе с использованием цикла `foreach` тип переменной «`i`» также заранее известен.

Для обобщенного списка перегружен индексатор, который позволяет получить элемент по его номеру в коллекции, элементы нумеруются с нуля. Пример:

```

int item2 = li[1];

//Работа с обобщенным списком
Console.WriteLine("\nОбобщенный (generic) список: ");
List<int> li = new List<int>();
li.Add(
void List<int>.Add(int item)
Добавляет объект в конец коллекции System.Collections.Generic.List<T>.
item: Объект, добавляемый в конец коллекции System.Collections.Generic.List<T>. Ссылочные типы данных поддерживают значение null.
{
    Console.WriteLine(i);
}

```

Рис. 19. Работа IntelliSense для добавления в обобщенный список.

Свойство Count возвращает количество элементов в коллекции:

```
int count = li.Count;
```

Метод Contains позволяет проверить существование элемента в коллекции, в данном примере проверяется существование в коллекции числа «3»:

```
bool is3 = li.Contains(3);
```

Метод Remove позволяет удалить элемент из коллекции, в данном примере из списка удаляется число «2»:

```
li.Remove(2);
```

Основное преимущество стандартных коллекций C# состоит в том, что для их обработки может быть использован интегрированный язык запросов LINQ. Это чрезвычайно облегчает обработку сложных данных в C#.

Ограничение рассмотренного примера состоит в том, что в такой список можно поместить только целые числа и нельзя поместить элементы других типов.

Если в список необходимо сохранять элементы различных типов, то можно использовать необобщенный список.

### 6.1.2 Необобщенный список

Рассмотрим пример работы с необобщенным списком. Создание объекта списка:

```
ArrayList al = new ArrayList();
```

Добавление элементов различных типов в список:

```
al.Add(333);
al.Add(123.123);
al.Add(«строка»);
```

Вывод элементов списка:

```
foreach (object o in al)
{
    //Для определения типа используется механизм рефлексии
    string type = o.GetType().Name;
    if (type == «Int32»)
    {
        Console.WriteLine(«Целое число: « + o.ToString());
    }
    else if (type == «String»)
    {
        Console.WriteLine(«Строка: « + o.ToString());
    }
    else
    {
        Console.WriteLine(«Другой тип: « + o.ToString());
    }
}
```

Преимуществом необобщенного списка является возможность добавления элементов различных типов.

Однако при чтении элементов возникает проблема, связанная с тем, что неизвестен тип элемента, который получен из списка, вернее это базовый тип `object`. Для решения этой проблемы может быть использован механизм рефлексии.

Механизм рефлексии рассматривается далее в пособии, здесь используется возможность получения информации о реальном типе элемента, получаемого из списка. Вызов метода «`o.GetType()`» для объекта возвращает информацию о его типе данных, свойство `Name` возвращает строковое имя типа. Далее осуществляется ветвление по строковому имени

типа, и выполняются различные действия в зависимости от различных значений типа данных.

### 6.1.3 Обобщенные стек и очередь

Стек и очередь отличаются от списка в первую очередь тем, что здесь не осуществляется перебор элементов коллекции, а осуществляется доступ к первому или последнему элементу коллекции. Принцип работы стека и очереди показан на рис 20.

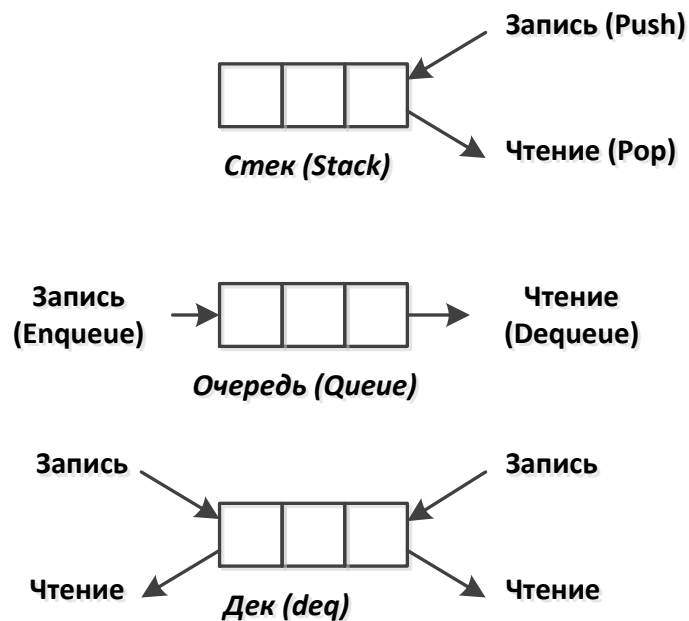


Рис. 20. Стек, очередь и дек.

В случае стека (класс `Stack`) работа происходит с так называемой вершиной стека, которую условно можно считать первым или последним элементом списка данных. Запись (`Push`) производится в вершину стека, чтение (`Pop`) также производится из вершины стека. При чтении элемент автоматически удаляется из стека. Стек реализует принцип работы LIFO (`last in, first out` – последний вошел, первый вышел).

В случае очереди (класс `Queue`) запись (`Enqueue`) производится в конец очереди, чтение (`Dequeue`) производится из начала очереди. При

чтении элемент автоматически удаляется из очереди. Очередь реализует принцип работы FIFO (first in, first out – первый вошел, первый вышел).

Необходимо отметить, что стек и очередь являются частным случаем структуры, которую принято называть дек (deq – double ended queue, двунаправленная очередь). В этом случае чтение и запись элементов может производиться с обоих концов списка данных. В .NET не существует стандартного библиотечного класса, реализующего дек.

Рассмотрим работу со стеком.

Создание стека:

```
Stack<int> st = new Stack<int>();
```

Добавление элементов в стек:

```
st.Push(1);
st.Push(2);
st.Push(3);
```

В реализации стека для .NET возможен перебор элементов с помощью foreach, хотя в классическом стеке такое действие не обязательно должно быть предусмотрено:

```
foreach(int i in st)
{
    Console.WriteLine(i);
}
```

С использованием метода Peek возможно чтение элемента из вершины стека без удаления (прочитанный элемент не удаляется из вершины стека):

```
int i3 = q.Peek();
```

С использованием метода Pop производится чтение элемента из вершины стека и его удаление:

```
while (st.Count > 0)
{
    int i = st.Pop();
    Console.WriteLine(i);
}
```

Рассмотрим работу с очередью.

Создание очереди:

```
Queue<int> q = new Queue<int>();
```



Добавление элементов в очередь:

```
q.Enqueue(11);
q.Enqueue(22);
q.Enqueue(33);
```

В реализации очереди для .NET возможен перебор элементов с помощью foreach:

```
foreach (int i in q)
{
    Console.WriteLine(i);
}
```

С использованием метода Peek возможно чтение элемента из начала очереди без удаления:

```
int i3 = q.Peek();
```

С использованием метода Dequeue производится чтение элемента из начала очереди и его удаление:

```
while (q.Count > 0)
{
    int i = q.Dequeue();
    Console.WriteLine(i);
}
```

Результаты вывода в консоль для стека и очереди:

Стек:

Перебор элементов стека с помощью foreach:

3

2

1

Получение верхнего элемента без удаления:3

Чтение с удалением элементов из стека:

3

2

1

Очередь:

Перебор элементов очереди с помощью foreach:

11

22

33

Получение первого элемента без удаления:11

Чтение с удалением элементов из очереди:

11

22

33

Обратите внимание, что чтение данных из стека производится в обратном порядке (в соответствии с принципом LIFO), а чтение данных из очереди в прямом порядке (в соответствии с принципом FIFO).

#### 6.1.4 Обобщенный словарь

Работа со словарем отличается от работы со списком в основном только тем, что в словарь добавляются пары ключ-значение.

В других языках программирования словарь могут также называть ассоциативным (или ассоциированным) массивом или хэш-таблицей.

Создание объекта словаря:

```
Dictionary<int, string> d = new Dictionary<int, string>();
```

При создании словаря указывается два обобщенных типа – тип ключа и тип значения.

Добавление элементов:

```
d.Add(1, "строка 1");
d.Add(2, "строка 2");
d.Add(3, "строка 3");
```

Вывод элементов:

```
foreach (KeyValuePair<int, string> v in d)
{
    Console.WriteLine(v.Key.ToString() + "-" + v.Value);
}
```

При выводе элементов возникает проблема, связанная с тем, что в цикле foreach необходимо объявить тип данных, который содержит пару ключ-значение, читаемые из словаря. В качестве такого типа в .NET

используется тип `KeyValuePair<TKey, TValue>`. Объект такого типа содержит свойства `Key` и `Value`, которые возвращают соответственно ключ и значение пары.

У объекта класса словаря существуют коллекции ключей и значений: `Keys` и `Values`. Пример вывода ключей и значений:

```
Console.WriteLine("\nКлючи: ");
foreach (int i in d.Keys)
{
    Console.WriteLine(i.ToString() + " ");
}

Console.WriteLine("\nЗначения: ");
foreach (string str in d.Values)
{
    Console.WriteLine(str + " ");
}
```

Для получения значения по ключу можно использовать перегруженный индексатор. Это аналогично использованию индексатора для списка. Пример:

```
int key = 3;
string val = d[key];
Console.WriteLine("\nДля ключа '" + key.ToString() +
    "' значение '" + val + "'");
```

Однако если указать ключ, который отсутствует в словаре, то это приведет к возникновению исключения `KeyNotFoundException`.

Для решения данной задачи можно также использовать метод `TryGetValue`, который не генерирует исключение, а возвращает логическое значение: `true` если удалось прочесть значение по ключу и `false` если не удалось. Пример:

```
string val2 = "";
bool res = d.TryGetValue(key, out val2);
if (res)
{
    Console.WriteLine("\nДля ключа '" + key.ToString()
        + "' значение '" + val2 + "'");
}
```

### 6.1.5 Кортеж

Кортеж – это сравнительно новый вид коллекции, который появился в .NET 4.0.

Кортеж является не совсем классической коллекцией. Если, например, список можно представить в виде таблицы из одного столбца и множества строк, а словарь в виде таблицы из двух столбцов и множества строк, то кортеж можно представить в виде таблицы из одной строки и множества столбцов.

Столбцы определяются динамически с помощью обобщенных типов. Пример объявления кортежа:

```
Tuple<int, string, string> group =
    new Tuple<int, string, string>(1, "ИУ", "ИУ-5");
```

Если лямбда-выражения позволяют «на лету» объявлять методы, то кортежи позволяют «на лету» объявлять структуры, подобные объектам классов.

Если программист хочет объявить какую-либо вспомогательную структуру данных и не хочет объявлять класс, то он может воспользоваться кортежем.

Однако, недостаток кортежей состоит в том, что поля кортежа в IntelliSense представляются не в виде имен (как в классе) а в виде номеров полей (Item1 ... ItemN). Пример представлен на рис. 21.

```
Console.WriteLine("\nКортежи");
Tuple<int, string, string> group = new Tuple<int, string, string>(1, "ИУ", "ИУ-5");
group.|
Console  Equals      group.ToString();
        GetHashCode
        GetType
//Клас  Item1      int Tuple<int, string, string>.Item1
//если  Item2      Получает значение первого компонента текущего объекта System.Tuple<T1,T2,T3>.
Tuple<  Item3
Console ToString    tuple = new
        Tuple<int, int, int, int, tuple<string, string, string>>(1, 2, 3, 4, new Tuple<string, string, string>("ИУ", "ИУ-5", "ИУ-5"));
```

Рис. 21. Работа IntelliSense для кортежа.

Кортеж не может содержать более 8 обобщенных типов, однако обобщенный тип может быть кортежем. Таким образом, если требуется использовать более 8 полей, то можно создавать вложенные кортежи.

Пример:

```
Tuple<int, int, int, int, int, int, int, Tuple<string, string, string>> tuple =
new Tuple<int, int, int, int, int, int, int, Tuple<string, string, string>>
(1, 2, 3, 4, 5, 6, 7, new Tuple<string, string, string>("str1", "str2", "str3"));
```

Кортеж может выступать в качестве обобщенного типа для других коллекций. Пример объявления списка, элементом которого является кортеж:

```
List<Tuple<int, int, int>> tupleList =
    new List<Tuple<int, int, int>>();
tupleList.Add(new Tuple<int, int, int>(1, 1, 1));
tupleList.Add(new Tuple<int, int, int>(2, 2, 2));
tupleList.Add(new Tuple<int, int, int>(3, 3, 3));
```

Вместо кортежа в данном примере можно было создать отдельный класс, содержащий три целочисленных поля, и помещать в список объекты данного класса.

### 6.1.6 Сортировка коллекций

Для большинства коллекций определен метод `Sort`, который выполняет сортировку коллекции. Если обобщенная коллекция основана на типе-значении то это не вызывает проблемы, так как правила упорядочивания элементов для типов-значений известны.

Пример сортировки коллекции для целых чисел:

```
Console.WriteLine("\nСортировка списка целых чисел:");
List<int> s1 = new List<int>();
s1.Add(5);
s1.Add(3);
s1.Add(2);
s1.Add(1);
s1.Add(4);
Console.WriteLine("\nПеред сортировкой:");
foreach (int i in s1) Console.Write(i.ToString() + " ");
//Сортировка
s1.Sort();
Console.WriteLine("\nПосле сортировки:");
foreach (int i in s1) Console.Write(i.ToString() + " ");
```

Результаты вывода в консоль:

Сортировка списка целых чисел:

Перед сортировкой:

5 3 2 1 4

После сортировки:

1 2 3 4 5

Однако в случае использования объектов ссылочного типа в качестве элементов коллекции возникает вопрос о том, как сортировать объекты сложных классов.

Рассмотрим дальнейшую работу с коллекциями на **примере 11**. Данный пример базируется на рассмотренных в примере 4 классах геометрических фигур.

Создание объектов классов фигур:

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

Добавление в список:

```
List<Figure> f1 = new List<Figure>();
f1.Add(circle);
f1.Add(rect);
f1.Add(square);
```

При вызове метода сортировки:

```
f1.Sort();
```

возникает исключение «InvalidOperationException» с сообщением «сбой при сравнении двух элементов массива».

Возникновение этого исключения связано с тем, что в классе геометрической фигуры и ее наследниках нет информации о том, как сравнить элементы, что необходимо для сортировки элементов.

Использование для коллекции метода Sort предполагает, что элементы коллекции реализуют интерфейс IComparable. Наследование от данного интерфейса предполагает реализацию метода CompareTo, который осуществляет сравнение двух объектов.

Реализуем данный метод для класса «Геометрическая фигура»:

```
public int CompareTo(object obj)
{
    //Приведение параметра к типу "фигура"
    Figure p = (Figure)obj;
    //Сравнение
    if (this.Area() < p.Area()) return -1;
    else if (this.Area() == p.Area()) return 0;
    else return 1; //(this.Area() > p.Area())
}
```

В данном методе предполагается, что производится сравнение двух параметров, которые стоят слева и справа от оператора сравнения.

Левым параметром является текущий объект класса (this), правым параметром является аргумент, передаваемый в метод (obj).

Вначале производится приведение obj к типу «Геометрическая фигура». Затем производится сравнение площади объекта this с площадью объекта obj. Если площадь this меньше, то метод возвращает «-1», если площади равны то «0», если площадь this больше, то «1». Правила формирования возвращаемых значений определяются интерфейсом IComparable, метод CompareTo возвращает «-1, 0, 1» потому что он должен возвращать такие значения в соответствии с требованиями разработчиков интерфейса IComparable.

Отметим, что метод Area является абстрактным и должен быть реализован в классах-наследниках. То есть в методе CompareTo задается правило сравнения элементов на основе площади, но правила вычисления площади задаются не в абстрактном классе «Геометрическая фигура», а в наследуемых от него классах.

Полный текст класса «Геометрическая фигура» с учетом реализации метода сравнения:

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

namespace FigureCollections
{
```

```

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

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

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

    /// <summary>
    /// Сравнение элементов (для сортировки)
    /// this - левый параметр сравнения
    /// </summary>
    /// <param name="obj">правый параметр сравнения</param>
    /// <returns>
    /// -1 - если левый параметр меньше правого
    /// 0 - параметры равны
    /// 1 - правый параметр меньше левого
    /// </returns>
    public int CompareTo(object obj)
    {
        //Приведение параметра к типу "фигура"
        Figure p = (Figure)obj;
        //Сравнение
        if (this.Area() < p.Area()) return -1;
        else if (this.Area() == p.Area()) return 0;
        else return 1; //(this.Area() > p.Area())
    }
}

```



```

    }
}

```

Обратите внимание, что наследование класса `Figure` от интерфейса `IComparable` является обязательным. Если просто реализовать метод `CompareTo` без наследования от интерфейса, то ошибка при сортировке все равно будет возникать. Можно сделать вывод, что метод `Sort` проверяет не просто наличие у класса элемента списка метода `CompareTo` (с помощью «утиной» типизации), но проверяет факт наследования класса элемента списка от интерфейса `IComparable`. В .NET такие проверки реализуются достаточно просто с использованием механизма рефлексии.

Отметим, что использование интерфейса `IComparable` имеет существенное ограничение – метод `CompareTo` может быть реализован единственным образом. То есть для класса данных можно задать только одну метрику сравнения, которая «встраивается» в класс данных. Например, в технологии LINQ для решения аналогичной задачи предложен более гибкий подход на основе классов-компараторов, которые реализуются «поверх» классов данных, что позволяет реализовать несколько классов-компараторов для одного класса данных.

После наследования класса `Figure` от интерфейса `IComparable` метод сортировки может быть вызван без возникновения исключений:

```

List<Figure> fl = new List<Figure>();
fl.Add(circle);
fl.Add(rect);
fl.Add(square);
Console.WriteLine("\nПеред сортировкой:");
foreach (var x in fl) Console.WriteLine(x);
//сортировка
fl.Sort();
Console.WriteLine("\nПосле сортировки:");
foreach (var x in fl) Console.WriteLine(x);

```

Результаты вывода в консоль:

Перед сортировкой:

Круг площадью 78,5398163397448

Прямоугольник площадью 20

Квадрат площадью 25

После сортировки:

Прямоугольник площадью 20

Квадрат площадью 25

Круг площадью 78,5398163397448

Таким образом, для сортировки коллекций, содержащих объекты классов, необходимо реализовывать в данных классах интерфейс `Comparable`.

## **6.2 Создание нестандартной коллекции на основе стандартной коллекции**

В том случае, когда необходимой стандартной коллекции в библиотеке не существует, прикладному программисту необходимо самостоятельно реализовывать требуемую коллекцию.

В этом случае существует два подхода:

1. Использовать стандартную коллекцию и написать «надстройку» над ней для реализации требуемого поведения. Данный путь является предпочтительным, так как требует относительно небольших ресурсов при кодировании.
2. В том случае, когда предыдущий подход невозможен, создавать коллекцию полностью «с нуля».

В данном разделе разбирается первый подход на примере класса «Разреженная матрица».

Разреженная матрица – тип данных, который достаточно часто используется в информатике. Это матрица большой размерности, которая может быть заполнена на 3-5%, остальные ячейки матрицы пусты. Использование в этом случае обычной матрицы совершенно нецелесообразно, так как потребует выделения очень больших объемов памяти для хранения пустых значений.

Разреженную матрицу можно реализовать «поверх» стандартной коллекции. В рассматриваемом примере разреженная матрица реализована на основе словаря. Ключом элемента словаря является комбинация индексов ячейки матрицы по строке и столбцу, значением элемента словаря является значение элемента матрицы.

Класс «Разреженная матрица» реализован в виде обобщенной коллекции, класс-обобщение T соответствует типу ячейки матрицы.

Полный текст класса «Разреженная матрица»:

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

namespace FigureCollections
{
    public class Matrix<T>
    {
        /// <summary>
        /// Словарь для хранения значений
        /// </summary>
        Dictionary<string, T> _matrix = new Dictionary<string, T>();

        /// <summary>
        /// Количество элементов по горизонтали (максимальное
        количество столбцов)
        /// </summary>
        int maxX;

        /// <summary>
        /// Количество элементов по вертикали (максимальное
        количество строк)
        /// </summary>
        int maxY;

        /// <summary>
        /// Реализация интерфейса для проверки пустого элемента
        /// </summary>
        IMatrixCheckEmpty<T> checkEmpty;

        /// <summary>
        /// Конструктор
        /// </summary>
        public Matrix(int px, int py,
                    IMatrixCheckEmpty<T> checkEmptyParam)
```

```

{
    this.maxX = px;
    this.maxY = py;
    this.checkEmpty = checkEmptyParam;
}

/// <summary>
/// Индексатор для доступа к данным
/// </summary>
public T this[int x, int y]
{
    set
    {
        CheckBounds(x, y);
        string key = DictKey(x, y);
        this._matrix.Add(key, value);
    }
    get
    {
        CheckBounds(x, y);
        string key = DictKey(x, y);
        if (this._matrix.ContainsKey(key))
        {
            return this._matrix[key];
        }
        else
        {
            return this.checkEmpty.getEmptyElement();
        }
    }
}

/// <summary>
/// Проверка границ
/// </summary>
void CheckBounds(int x, int y)
{
    if (x < 0 || x >= this.maxX)
    {
        throw new ArgumentOutOfRangeException("x",
            "x=" + x + " выходит за границы");
    }
    if (y < 0 || y >= this.maxY)
    {
        throw new ArgumentOutOfRangeException("y",
            "y=" + y + " выходит за границы");
    }
}

/// <summary>

```

```

    /// Формирование ключа
    /// </summary>
    string DictKey(int x, int y)
    {
        return x.ToString() + "_" + y.ToString();
    }

    /// <summary>
    /// Приведение к строке
    /// </summary>
    /// <returns></returns>
    public override string ToString()
    {
        StringBuilder b = new StringBuilder();
        for (int j = 0; j < this.maxY; j++)
        {
            b.Append("[");
            for (int i = 0; i < this.maxX; i++)
            {
                //Добавление разделителя-табуляции
                if (i > 0)
                {
                    b.Append("\t");
                }
                //Если текущий элемент не пустой
                if (!this.checkEmpty.checkEmptyElement(this[i, j]))
                {
                    //Добавить приведенный к строке текущий элемент
                    b.Append(this[i, j].ToString());
                }
                else
                {
                    //Иначе добавить признак пустого значения
                    b.Append(" - ");
                }
            }
            b.Append("]\n");
        }
        return b.ToString();
    }
}

```

Рассмотрим работу данного класса более подробно.

Основной структурой данных для хранения разреженной матрицы является словарь «\_matrix». Ключом словаря является строка (которая содержит комбинацию индексов ячейки матрицы по строке и столбцу),

значением словаря является обобщенный тип T (который является типом значения элемента матрицы).

Поля данных maxX и maxY используются для хранения размеров матрицы.

Поле «checkEmpty» содержит объект класса, реализующего интерфейс «IMatrixCheckEmpty<T>» для работы с пустыми значениями.

Рассмотрим данный интерфейс более подробно:

```
using System;

namespace FigureCollections
{
    /// <summary>
    /// Проверка пустого элемента матрицы
    /// </summary>
    public interface IMatrixCheckEmpty<T>
    {
        /// <summary>
        /// Возвращает пустой элемент
        /// </summary>
        T getEmptyElement();

        /// <summary>
        /// Проверка что элемент является пустым
        /// </summary>
        bool checkEmptyElement(T element);
    }
}
```

Интерфейс содержит два метода:

- getEmptyElement, который возвращает пустой элемент разреженной матрицы;
- checkEmptyElement, который проверяет, что переданное в качестве параметра метода значение является пустым элементом матрицы.

Методы данного интерфейса используются при реализации разреженной матрицы.

Вернемся к классу разреженной матрицы Matrix<T>.

Класс содержит конструктор, в который в качестве параметров передаются размеры матрицы по горизонтали и вертикали и объект класса, реализующего интерфейс `IMatrixCheckEmpty<T>`. Конструктор сохраняет параметры в соответствующих полях класса.

Наиболее важной частью реализации является индексатор `«public T this[int x, int y]»`. В качестве параметров индексатору передаются координаты `x` и `y` – столбец и строка текущей ячейки матрицы.

`Set`-аксессор индексатора осуществляет запись элемента в матрицу.

Вначале осуществляется проверка границ матрицы с помощью метода `CheckBounds`. Данный метод проверяет, что координаты текущей ячейки `x` и `y` находятся в диапазоне от 0 до границы по `x` или `y` соответственно. В случае выхода за границы генерируется исключение `ArgumentOutOfRangeException` с соответствующим сообщением.

Если координаты текущей ячейки находятся в пределах требуемых границ, то вычисляется ключ для записи в словарь с помощью функции `DictKey`. Данная функция осуществляет строковую конкатенацию координат `x` и `y` ячейки. Отметим, что функция `DictKey` является устойчивой – всегда возвращает одинаковый результат для одного и того же набора параметров. Поэтому данная функция может быть использована как для записи, так и для чтения данных.

После вычисления ключа производится запись значения, присваиваемого в `Set`-аксесоре (`value`), в словарь `«_matrix»`. В данной реализации не проверяется, существует ли уже в словаре ячейка с таким ключом, как и в случае обычного массива, элементы перезаписываются. Однако, в случае необходимости, можно добавить дополнительную проверку на наличие элемента в словаре и запретить перезапись элементов, при попытке перезаписи может генерироваться исключение.

`Get`-аксессор индексатора осуществляет чтение элемента из матрицы.

Как и в случае записи, при чтении сначала осуществляется проверка границ матрицы с помощью метода `CheckBounds` и если координаты текущей ячейки находятся в пределах требуемых границ, то вычисляется ключ для чтения из словаря с помощью функции `DictKey`.

Далее осуществляется проверка существования ключа в словаре с помощью метода «`_matrix.ContainsKey`». Если элемент с вычисленным ключом существует, то `Get`-аксессор возвращает значение элемента. Если элемент не существует, то возвращается пустое значение с использованием поля `checkEmpty`.

В классе разреженной матрицы также переопределен метод `ToString`, который выводит значение матрицы в строковом представлении. Алгоритм работы данного метода является обычным, в нем с использованием двух вложенных циклов осуществляется перебор и вывод в строку ячеек матрицы.

Для работы со строками в данном методе используется класс `StringBuilder`. Зачем необходим этот класс, ведь можно создавать объекты класса `String` путем их конкатенации с помощью оператора «`+`»? Но класс `String` содержит неизменяемые строки, использованием оператора «`+`» приводит к порождению новых объектов, что плохо сказывается на производительности приложения.

Если класс `String` и оператор «`+`» используются в небольших фрагментах кода, то это не составляет проблемы. Но если этот способ активно использовать в приложении, например для формирования больших текстовых отчетов, то будет создаваться большое количество ненужных строковых объектов, которые практически сразу должен будет удалить сборщик мусора. Разработчики платформы `.NET` предупреждают, что это может существенно снизить производительность приложения, поэтому в таком случае необходимо использовать класс `StringBuilder`.



Класс `StringBuilder` позволяет формировать сложные строки без возникновения проблем с производительностью. У класса `StringBuilder` существует метод `Append`, который позволяет дописать в конец внутренней строки `StringBuilder` практически любые данные. Также для класса `StringBuilder` перегружен метод `ToString`, который возвращает внутреннюю строку `StringBuilder` в виде объекта класса `String`.

Рассмотрим пример использования класса «Разреженная матрица». Вначале необходимо создать класс, реализующий интерфейс `IMatrixCheckEmpty`:

```
using System;

namespace FigureCollections
{
    class FigureMatrixCheckEmpty : IMatrixCheckEmpty<Figure>
    {
        /// <summary>
        /// В качестве пустого элемента возвращается null
        /// </summary>
        public Figure getEmptyElement()
        {
            return null;
        }

        /// <summary>
        /// Проверка что переданный параметр равен null
        /// </summary>
        public bool checkEmptyElement(Figure element)
        {
            bool Result = false;
            if (element == null)
            {
                Result = true;
            }
            return Result;
        }
    }
}
```

Пример использования объекта класса «Разреженная матрица»:

```
Console.WriteLine("\nМатрица");
Matrix<Figure> matrix = new Matrix<Figure>(3, 3,
                                         new FigureMatrixCheckEmpty());
matrix[0, 0] = rect;
```

```
matrix[1, 1] = square;
matrix[2, 2] = circle;
Console.WriteLine(matrix.ToString());
```

Результаты вывода в консоль:

Матрица

```
[Прямоугольник площадью 20      -      - ]
[ -      Квадрат площадью 25      - ]
[ -      -      Круг площадью 78,5398163397448]
```

Таким образом, память в разреженной матрице выделяется только для реально существующих элементов. Пустые элементы не хранятся в памяти, вместо них возвращается заданное по умолчанию пустое значение.

### **6.3 Создание нестандартной коллекции без использования стандартных коллекций**

В данном разделе разбирается пример создания коллекции в том случае, когда использование стандартных коллекций нецелесообразно. Например, это реализация каких-либо сложных деревьев, что достаточно трудно сделать на основе стандартных коллекций.

Однако реализация дерева сама по себе является непростой задачей. Поэтому в данном разделе в качестве учебного примера создается простой однонаправленный список и в качестве его наследника реализуется класс стека. Данные классы создаются полностью «с нуля» без использования стандартных коллекций.

Рассматриваемые в данном разделе классы, как и классы предыдущего раздела, находятся в **примере 11**.

#### **6.3.1 Классы простого списка**

Класс SimpleListItem является контейнером элемента списка:

```
using System;

namespace FigureCollections
{
    /// <summary>
```

```

/// Элемент списка
/// </summary>
public class SimpleListItem<T>
{
    /// <summary>
    /// Данные
    /// </summary>
    public T data { get; set; }

    /// <summary>
    /// Следующий элемент
    /// </summary>
    public SimpleListItem<T> next { get; set; }

    ///конструктор
    public SimpleListItem(T param)
    {
        this.data = param;
    }
}
}

```

Типом-обобщением `T` является тип данных списка. Класс `SimpleListItem` содержит свойство `data` обобщенного типа, которое предназначено для хранения данных.

Класс `SimpleListItem` также содержит свойство `next` типа `SimpleListItem`, которое является аналогом указателя на следующий элемент. Данное объявление может показаться несколько странным по сравнению с C++. В C++ при создании собственных структур данных используются указатели. Здесь свойство `next` кажется скорее вложенным в предыдущий элемент, так что получается своеобразная «матрешка» из вложенных элементов списка. Однако синтаксис в этом случае не совсем точно отражает работу с памятью. Классы в .NET являются ссылочными типами, поэтому при объявлении переменной типа объект класса в данной переменной будет храниться именно ссылка на объект класса, то есть фактически указатель. Поэтому в памяти последовательность объектов класса `SimpleListItem` будет представлена в виде цепочки ссылок.

Класс `SimpleList` реализует список:

```
using System;
```

```

using System.Collections.Generic;

namespace FigureCollections
{
    /// <summary>
    /// Список
    /// </summary>
    public class SimpleList<T> : IEnumerable<T>
        where T : IComparable
    {
        /// <summary>
        /// Первый элемент списка
        /// </summary>
        protected SimpleListItem<T> first = null;

        /// <summary>
        /// Последний элемент списка
        /// </summary>
        protected SimpleListItem<T> last = null;

        /// <summary>
        /// Количество элементов
        /// </summary>
        public int Count
        {
            get { return _count; }
            protected set { _count = value; }
        }
        int _count;

        /// <summary>
        /// Добавление элемента
        /// </summary>
        public void Add(T element)
        {
            SimpleListItem<T> newItem =
                new SimpleListItem<T>(element);
            this.Count++;

            //Добавление первого элемента
            if (last == null)
            {
                this.first = newItem;
                this.last = newItem;
            }
            //Добавление следующих элементов
            else
            {
                //Присоединение элемента к цепочке
                this.last.next = newItem;
            }
        }
    }
}

```

```

        //Присоединенный элемент считается последним
        this.last = newItem;
    }
}

/// <summary>
/// Чтение контейнера с заданным номером
/// </summary>
public SimpleListItem<T> GetItem(int number)
{
    if ((number < 0) || (number >= this.Count))
    {
        //Можно создать собственный класс исключения
        throw new Exception("Выход за границу индекса");
    }
    SimpleListItem<T> current = this.first;
    int i = 0;
    //Пропускаем нужное количество элементов
    while (i < number)
    {
        //Переход к следующему элементу
        current = current.next;
        //Увеличение счетчика
        i++;
    }
    return current;
}

/// <summary>
/// Чтение элемента с заданным номером
/// </summary>
public T Get(int number)
{
    return GetItem(number).data;
}

/// <summary>
/// Для перебора коллекции
/// </summary>
public IEnumerator<T> GetEnumerator()
{
    SimpleListItem<T> current = this.first;

    //Перебор элементов
    while (current != null)
    {
        //Возврат текущего значения
        yield return current.data;
        //Переход к следующему элементу
        current = current.next;
    }
}

```

```

    }
}

//Реализация обобщенного IEnumerator<T> требует реализации
необобщенного интерфейса
//Данный метод добавляется автоматически при реализации интерфейса
System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

/// <summary>
/// Сортировка
/// </summary>
public void Sort()
{
    Sort(0, this.Count - 1);
}

/// <summary>
/// Алгоритм быстрой сортировки
/// </summary>
private void Sort(int low, int high)
{
    int i = low;
    int j = high;
    T x = Get((low + high) / 2);
    do
    {
        while (Get(i).CompareTo(x) < 0) ++i;
        while (Get(j).CompareTo(x) > 0) --j;
        if (i <= j)
        {
            Swap(i, j);
            i++; j--;
        }
    } while (i <= j);

    if (low < j) Sort(low, j);
    if (i < high) Sort(i, high);
}

/// <summary>
/// Вспомогательный метод для обмена элементов при
сортировке
/// </summary>
private void Swap(int i, int j)
{
    SimpleListItem<T> ci = GetItem(i);

```

```

        SimpleListItem<T> cj = GetItem(j);
        T temp = ci.data;
        ci.data = cj.data;
        cj.data = temp;
    }
}

```

Рассмотрим сначала поля и методы, которые отвечают за внутреннее устройство списка.

Поля `first` и `last` содержат контейнеры (вернее, ссылки на контейнеры) для первого и последнего элементов списка.

Свойство `Count` содержит количество элементов. Оно доступно для чтения везде, в том числе и снаружи класса (область видимости `public`), а для записи только в классе и его наследниках (область видимости `protected`).

Метод добавления нового элемента в конец списка `Add` создает новый контейнер элемента на основе переданных данных (объект класса `SimpleListItem`), увеличивает количество элементов списка на 1, затем добавляет контейнер к цепочке контейнеров. В результате добавления элемента поле `first` содержит ссылку на первый, а `last` на последний элементы списка.

Метод `GetItem` используется для получения контейнера по его порядковому номеру. Для получения контейнера с номером `N` метод перебирает в цикле `N-1` контейнеров и возвращает `N`-ый контейнер. Метод `Get` возвращает данные, находящиеся в контейнере.

Рассмотренные свойства и методы являются основными для работы списка. Также у класса списка реализована дополнительная функциональность.

Данные списка можно перебирать в цикле `foreach`. Для того чтобы в .NET элементы коллекции можно перебирать в цикле `foreach`, класс коллекции должен реализовывать интерфейс `IEnumerable`. Класс `SimpleList<T>` реализует обобщенный интерфейс `IEnumerable<T>`.

Реализация данного интерфейса требует реализации у класса методов `GetEnumerator`.

Обобщенный метод `«public IEnumerator<T> GetEnumerator()»` осуществляет перебор всех элементов списка и их возврат с помощью конструкции `«yield return»`. Реализация обобщенного интерфейса `IEnumerator<T>` требует реализации необобщенного интерфейса `IEnumerator`. Поэтому класс `SimpleList` также реализует необобщенный метод `«System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()»`, который вызывает обобщенную реализацию `GetEnumerator`.

Также для простого списка реализована сортировка методом быстрой сортировки (`Quick Sort`). Метод `Sort` без параметров осуществляет сортировку, метод `Sort` с параметрами используется для внутренних рекурсивных вызовов. Вспомогательный метод `Swap` используется для перестановки элементов.

Напомним, что алгоритм быстрой сортировки был разработан Чарльзом Хоаром во время его стажировки в Московском Государственном Университете в 1960 году. Интересно, что алгоритм изначально разрабатывался для сортировки данных на магнитной ленте (устройстве с последовательным доступом), однако, в результате оказался одним из самых эффективных и для сортировки коллекций с произвольным доступом к элементам.

Идея алгоритма состоит в том, что выбирается некоторый средний элемент коллекции и предполагается, что он находится на своем месте. Далее перебираются все элементы коллекции слева и справа от элемента. Если слева находится больший элемент, а справа меньший, то они меняются местами. В результате такого прохода выбранный элемент действительно находится на своем месте, слева от него находятся все меньшие, а справа все большие элементы, однако элементы справа и слева



не упорядочены. Поэтому далее алгоритм вызывается отдельно для левой и правой части, где также выбираются средние элементы и работа алгоритма рекурсивно повторяется до тех пор, пока размер сортируемого подмассива не будет равен одному элементу.

Не смотря на «наивность» и относительную простоту реализации алгоритма, доказано, что он является одним из самых эффективных алгоритмов сортировки. Одной из проблем алгоритма является большое количество рекурсивных вызовов, что может переполнить стек вызовов. Поэтому в некоторых реализациях данного алгоритма рекурсию заменяют на цикл, а вместо стека вызовов используют отдельный стек данных, в котором хранят индексы начала и конца сортируемого подмассива.

Рассмотрим пример использования класса SimpleList:

```
SimpleList<Figure> list = new SimpleList<Figure>();
list.Add(circle);
list.Add(rect);
list.Add(square);
Console.WriteLine("\nПеред сортировкой:");
foreach (var x in list) Console.WriteLine(x);
//сортировка
list.Sort();
Console.WriteLine("\nПосле сортировки:");
foreach (var x in list) Console.WriteLine(x);
```

В данном примере метод Add используется для добавления данных в список. Использование списка в цикле foreach приводит к вызову обобщенного метода GetEnumerator. Метод Sort используется для сортировки данных списка.

Результаты вывода в консоль:

Список

Перед сортировкой:

Круг площадью 78,5398163397448

Прямоугольник площадью 20

Квадрат площадью 25

После сортировки:

Прямоугольник площадью 20

Квадрат площадью 25

Круг площадью 78,5398163397448

### 6.3.2 Класс стека

Класс стека наследуется от класса списка:

```
using System;

namespace FigureCollections
{
    /// <summary>
    /// Класс стек
    /// </summary>
    class SimpleStack<T> : SimpleList<T> where T : IComparable
    {
        /// <summary>
        /// Добавление в стек
        /// </summary>
        public void Push(T element)
        {
            //Добавление в конец списка уже реализовано
            Add(element);
        }

        /// <summary>
        /// Удаление и чтение из стека
        /// </summary>
        public T Pop()
        {
            //default(T) - значение для типа T по умолчанию
            T Result = default(T);
            //Если стек пуст, возвращается значение по умолчанию для типа
            if (this.Count == 0) return Result;
            //Если элемент единственный
            if (this.Count == 1)
            {
                //то из него читаются данные
                Result = this.first.data;
                //обнуляются указатели начала и конца списка
                this.first = null;
                this.last = null;
            }
            //В списке более одного элемента
            else
            {
                //Поиск предпоследнего элемента
                SimpleListItem<T> newLast = this.GetItem(this.Count - 2);
                //Чтение значения из последнего элемента
                Result = newLast.next.data;
                //предпоследний элемент считается последним
            }
        }
    }
}
```

```

        this.last = newLast;
        //последний элемент удаляется из списка
        newLast.next = null;
    }
    //Уменьшение количества элементов в списке
    this.Count--;
    //Возврат результата
    return Result;
}
}
}

```

Класс стека наследуют от списка все необходимые методы за исключением собственно методов стека – Push (запись в стек) и Pop (чтение с удалением из стека).

При реализации стека на основе списка необходимо решить, что будет являться вершиной стека – начало или конец списка. В данной реализации в качестве вершины стека используется конец списка.

Метод Push просто вызывает метод Add для добавления данных в конец списка.

Метод Pop в готовом виде в списке не реализован. Метод Pop возвращает последний элемент списка и удаляет его из списка. Реализация метода зависит от количества элементов в списке. Если список не содержит элементов, то возвращается значение по умолчанию для обобщенного типа «default(T)». Если список содержит один элемент, то он возвращается, а список переводится в состояние пустого списка. Если список содержит более одного элемента, то возвращается и удаляется из списка последний элемент, а предпоследний элемент устанавливается в качестве последнего.

Рассмотрим пример использования класса SimpleStack:

```

SimpleStack<Figure> stack = new SimpleStack<Figure>();
//добавление данных в стек
stack.Push(rect);
stack.Push(square);
stack.Push(circle);
//чтение данных из стека
while (stack.Count > 0)
{
    Figure f = stack.Pop();
}

```

```

    Console.WriteLine(f);
}

```

Результаты вывода в консоль:

Стек

Круг площадью 78,5398163397448

Квадрат площадью 25

Прямоугольник площадью 20

В соответствии с ожидаемым алгоритмом работы метод Pop извлекает элементы из вершины стека в обратном порядке.

## 7 Работа с файловой системой

Платформа .NET и язык C# предоставляют широкие возможности для работы с файловой системой. В данном разделе рассматриваются некоторые из этих возможностей на основе **примера 12**.

### 7.1 Получение данных о файлах и каталогах

Наиболее простым способом получения информации о файловой системе является использование статического класса Directory. Если класс является статическим, то предполагается, что все его свойства и методы являются статическими.

Класс Directory, объявленный в пространстве имен System.IO, позволяет осуществлять работу с каталогами: создание (метод CreateDirectory), удаление (метод Delete), перемещение (метод Move) каталогов, получение списка файлов и подкаталогов.

В следующем примере осуществляется вывод списка файлов и каталогов в корне диска «C»:

```

string catalogName = @"c:\";
Console.WriteLine("\nСписок файлов каталога " + catalogName);
string[] files = Directory.GetFiles(catalogName);
foreach (string file in files)
{
    Console.WriteLine(file);
}

```

```

Console.WriteLine("\nСписок подкаталогов каталога " + catalogName);
string[] dirs = Directory.GetDirectories(catalogName);
foreach (string dir in dirs)
{
    Console.WriteLine(dir);
}

```

Метод «Directory.GetFiles» получает список файлов, а метод «Directory.GetDirectories» получает список подкаталогов указанного каталога.

Обратите внимание на объявление переменной `catalogName`. Как и в C++ в C# символ «\» внутри строки используется для создания специальных символов, например «\n» – перевод строки. В соответствии с правилами C# имя каталога должно быть объявлено как «с:\», как и в C++ символ «\» удваивается. Но особенностью C# является то, что если перед строковым значением стоит символ «@», то специальные символы игнорируются и удваивать символ «\» не требуется.

Результаты вывода в консоль:

```

Список файлов каталога c:\
c:\AVScanner.ini
c:\pagefile.sys
c:\SecurityScanner.dll

```

```

Список подкаталогов каталога c:\
c:\$Recycle.Bin
c:\apps
c:\Documents and Settings
c:\Program Files
c:\Program Files (x86)
c:\Temp
c:\Windows

```

## 7.2 Чтение и запись текстовых файлов

В данном разделе рассматривается наиболее простой способ работы с текстовыми файлами на основе класса File.

Класс File обладает достаточно широкой функциональностью. С его помощью возможно копирование (метод Copy), перемещение (метод Move), удаление (метод Delete) файлов в файловой системе.

У класса файл существует достаточно большое количество методов для работы с файлами на основе потоков (класс FileStream). Класс FileStream позволяет выполнять с файлами данных большое количество действий, однако данный подход является достаточно низкоуровневым и в пособии детально не рассматривается. Данный подход детально рассмотрен в [1, 2]. Отметим, что данный подход пришел в C# из стандартных библиотек Java.

Для работы с бинарными файлами у класса File существуют упрощенные по сравнению с потоками методы чтения (ReadAllBytes) и записи (WriteAllBytes). Данные методы можно применять для обработки последовательности байтов. Данные методы также можно применять для бинарной сериализации/десериализации объектов, однако для этой задачи существуют более эффективные классы, рассмотренные в следующем разделе.

Рассмотрим более детально методы чтения и записи текстовых файлов, упрощенные по сравнению с потоковыми методами.

Метод File.ReadAllText возвращает считанный текстовый файл в виде переменной типа string. Метод File.WriteAllText записывает переменную типа string в качестве содержимого текстового файла.

Для дальнейшей обработки текстовых файлов необходимо получать путь к файлу, который располагается в каталоге рядом с исполняемой программой:

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

```

/// Возвращает путь к текущему исполняемому файлу
/// </summary>
static string GetExecPath()
{
    //Получение пути и имени текущего исполняемого файла
    //с помощью механизма рефлексии
    string exeFileName =
System.Reflection.Assembly.GetExecutingAssembly().Location;
    //Получение пути к текущему исполняемому файлу
    string Result = Path.GetDirectoryName(exeFileName);
    return Result;
}

```

Имя исполняемого файла получается с использованием механизма рефлексии как имя текущей исполняемой сборки – «System.Reflection.Assembly.GetExecutingAssembly().Location». Далее метод GetDirectoryName класса Path из полного имени каталога и файла возвращает только имя каталога. Класс Path используется для действий с именами каталогов и файлов.

Пример ввода строки и сохранения в файл:

```

//Текущий каталог
string currentPath = GetExecPath();
Console.WriteLine("Введите текст для записи в файл:");
//Ввод содержимого файла
string file1Contents = Console.ReadLine();
//Формирование имени файла
string file1Name = Path.Combine(currentPath, "file1.txt");
//Запись строки в файл
File.WriteAllText(file1Name, file1Contents);
//Чтение строки из файла
string file1ContentsRead = File.ReadAllText(file1Name);
Console.WriteLine("Чтение строки из файла:");
Console.WriteLine(file1ContentsRead);

```

В данном примере содержимое строки вводится с клавиатуры, затем с помощью метода File.WriteAllText записывается в текстовый файл. Для формирования имени файла используется метод Path.Combine, который корректно соединяет строку пути к текущему исполняемому файлу и имя создаваемого текстового файла. Если указанный текстовый файл существует, то метод File.WriteAllText удаляет предыдущее содержимое файла. Метод File.ReadAllText возвращает считанный текстовый файл в

виде строки. В данном примере файл содержит только одну строку, однако метод `File.ReadAllText` может считать текстовый файл большого размера, содержащий произвольное количество переводов строк.

Результаты вывода в консоль:

Введите текст для записи в файл: пример текста

Чтение строки из файла: пример текста

При этом в каталоге, содержащем исполняемый файл, действительно создается текстовый файл `file1.txt` с указанным содержимым.

Класс `File` также содержит методы, которые могут записать (`WriteAllLines`) или прочитать (`ReadAllLines`) массив строк.

Рассмотрим пример использования данных методов:

```

Console.WriteLine("Введите строки для записи в файл (пустая строка -
окончание ввода):");
string tempStrTrim = "";
//Список строк
List<string> list = new List<string>();
do
{
    //Временная переменная для хранения строки
    string tempStr = Console.ReadLine();
    //Удаление пробелов из введенной строки
    tempStrTrim = tempStr.Trim();
    if (tempStrTrim != "")
    {
        //Непустая строка сохраняется в список
        list.Add(tempStrTrim);
    }
}
while (tempStrTrim != "");

//Формирование имени файла
string file2Name = Path.Combine(currentPath, "file2.txt");
//Запись в файл массива строк
File.WriteAllLines(file2Name, list.ToArray());

//Чтение строк из файла
string[] file2ContentsRead = File.ReadAllLines(file2Name);
Console.WriteLine("Чтение строк из файла:");
foreach (string str in file2ContentsRead)
{
    Console.WriteLine(str);
}

```



Для хранения строк при вводе используется обобщенный список. Ввод строк осуществляется с помощью цикла с постусловием, в условии проверяется ввод пустой строки, то есть ввод строк осуществляется до ввода пустой строки. Для введенной строки с использованием метода Trim удаляются пробелы в начале и конце строки. Если строка не является пустой, то она добавляется в список.

Отметим, что в данном примере для проверки пустоты строки выполняется сравнение с пустой строкой (`tempStrTrim != ""`). Данный метод работает корректно с учетом того что для переменной `tempStrTrim` с использованием метода Trim предварительно удалены пробелы в начале и конце строки. Но для решения данной задачи в .NET также существует статический метод класса `string` – «`String.IsNullOrEmpty(строка)`», который возвращает истину, если строка является пустой или содержит только пробелы.

Для записи в файл массива строк используется метод `File.WriteAllLines`, список строк преобразуется в массив с использованием метода `list.ToArray`.

Для чтения из файла массива строк используется метод `File.ReadAllLines`, для вывода прочитанных строк используется цикл `foreach`.

Результаты вывода в консоль:

Введите строки для записи в файл (пустая строка - окончание ввода):

строка1

строка2

строка3

Чтение строк из файла:

строка1

строка2

строка3

При этом в каталоге, содержащем исполняемый файл, действительно создается текстовый файл file2.txt с указанным содержимым.

Отметим, что на практике для чтения текстового файла обычно используют метод `File.ReadAllText`, а далее текстовый файл, считанный в виде строки, делят на подстроки с использованием метода `string.Split`. Метод `Split` позволяет разделить строку на подстроки с использованием произвольного разделителя, которым может быть символ переноса строки (в этом случае получается массив строк текстового файла) или пробел (в этом случае получается массив слов текстового файла).

### **7.3 Сериализация и десериализация объектов**

Одной из наиболее частых задач, которая возникает при работе с файлами, является задача сохранения в файл или чтения из файла объектов классов.

Задача сохранения объекта называется сериализацией, а задача чтения объекта называется десериализацией. В результате сериализации получается «твердая копия» объекта в виде файла. Эта копия может быть куда-либо сохранена, перемещена и т.д. В результате десериализации объект восстанавливается на основе «твердой копии» и с ним можно продолжить работу.

Отметим, что в настоящее время наиболее часто для сериализации/десериализации используется база данных. В .NET существует технология Entity Framework, которая обеспечивает объектно-реляционное отображение (ORM – object-relational mapping) между объектами классов C# и записями в таблице реляционной базы данных. Фактически Entity Framework решает задачу сериализации/десериализации объектов классов C# в таблицы реляционной БД. Также Entity Framework

решает множество других задач, в том числе облегчение разработки запросов к данным с помощью технологии LINQ to Entities.

В данном разделе на основе **примера 13** рассматриваются наиболее простые методы сериализации/десериализации объектов в файлы.

Классы, используемые для данной задачи, расположены в пространстве имен System.Runtime.Serialization.

### 7.3.1 Бинарная сериализация/десериализация

Наиболее часто используется бинарная сериализация/десериализация с использованием класса BinaryFormatter.

Вначале создадим класс данных, для которого будет использована бинарная сериализация/десериализация:

```
using System;

namespace Serialization
{
    [Serializable]
    class DataBin
    {
        public int int1 { get; set; }
        public double double1 { get; set; }
        public string str1 { get; set; }

        public override string ToString()
        {
            return
                "str1=" + str1
                + " int1=" + int1.ToString()
                + " double1=" + double1.ToString();
        }
    }
}
```

Класс содержит три свойства строкового, целого и вещественного типов. Для класса перегружен метод ToString который выводит значения свойств в строковом формате. Для использования бинарной сериализации класс обязательно должен быть помечен атрибутом «[Serializable]» (использование атрибутов разбирается далее в разделе рефлексии).

Далее создадим объект класса:

```
DataBin b = new DataBin()
{
    str1 = "строка1",
    int1 = 333,
    double1 = 123.45
};
```

Обратите внимание, что объект класса создается не совсем обычным способом: вначале вызывается пустой конструктор класса DataBin, а после этого ставятся фигурные скобки и происходит инициализация свойств класса, свойства разделяются запятыми. Такой синтаксис создания объектов класса допустим начиная с .NET 4.0. Отметим, что при такой форме инициализации пустые скобки при вызове конструктора можно не ставить. Следующий пример также является корректным:

```
DataBin b = new DataBin
{
    str1 = "строка1",
    int1 = 333,
    double1 = 123.45
};
```

Рассмотрим пример сериализации объекта:

```
Console.WriteLine("До сериализации:");
Console.WriteLine(b);
//Текущий каталог
string currentPath = GetExecPath();
//Формирование имени файла
string fileBinName = Path.Combine(currentPath, "file1.bin");
//При сериализации файл необходимо открыть с использованием потоков
Stream TestFileStream1 = File.Create(fileBinName);
//Создание объекта класса сериализации
BinaryFormatter serializer = new BinaryFormatter();
//Сериализация объекта в файл
serializer.Serialize(TestFileStream1, b);
//Закрытие потока
TestFileStream1.Close();
```

Вначале осуществляется вывод в консоль сериализуемого объекта в строковом виде. Далее формируется имя файла для сериализации в переменной fileBinName.

Для сериализации необходимо открывать файл с использованием потоков. Файл открывается в режиме записи с использованием метода `File.Create`. После завершения работы с потоком необходимо закрыть поток с использованием метода `Close`.

Сериализация и десериализация осуществляются с использованием объекта `serializer` класса `BinaryFormatter`. Для сериализации используется метод `Serialize`, которому в качестве параметров передаются открытый поток для записи и сериализуемый объект.

Рассмотрим пример десериализации объекта:

```
//Открытие файла в виде потока на чтение
Stream TestFileStream2 = File.OpenRead(fileBinName);
//Десериализация объекта
DataBin b2 = (DataBin)serializer.Deserialize(TestFileStream2);
//Закрытие потока
TestFileStream2.Close();
Console.WriteLine("После десериализации:");
Console.WriteLine(b2);
```

Для десериализации также необходимо открывать файл с использованием потоков. Файл открывается в режиме чтения с использованием метода `File.OpenRead`. После завершения работы с потоком необходимо закрыть поток с использованием метода `Close`.

Для десериализации используется метод `Deserialize` класса `BinaryFormatter`, которому в качестве параметра передается открытый поток. Метод возвращает десериализованный объект в виде объекта класса `Object`, далее необходимо осуществить приведение объекта класса `Object` к нужному типу, в данном примере это приведение к типу `DataBin`.

Результаты вывода в консоль:

Бинарная сериализация/десериализация

До сериализации:

str1=строка1 int1=333 double1=123,45

После десериализации:

str1=строка1 int1=333 double1=123,45

Таким образом, объект сериализован в бинарный файл, затем десериализован из бинарного файла. Восстановленный объект полностью совпадает с исходным.

### 7.3.2 Сериализация/десериализация в формат XML

Данный вид сериализации/десериализации также используется достаточно часто. Принцип работы практически полностью совпадает в бинарной сериализацией/десериализацией, только вместо класса BinaryFormatter используется класс XmlSerializer.

Рассмотрим пример сериализации/десериализации. Класс данных:

```
using System;

namespace Serialization
{
    public class DataXml1
    {
        public int int1 { get; set; }
        public double double1 { get; set; }
        public string str1 { get; set; }

        public override string ToString()
        {
            return
                "str1=" + str1
                + " int1=" + int1.ToString()
                + " double1=" + double1.ToString();
        }
    }
}
```

В случае использования XML-сериализации атрибут «[Serializable]» не требуется.

Создание объекта класса данных, сериализация/десериализация:

```
Console.WriteLine("Сериализация/десериализация в формате XML:");

DataXml1 dataXml1 = new DataXml1
{
    str1 = "строка1 xml",
    int1 = 3333,
    double1 = 333.33
};
```

```

//+++++
//сериализация
//+++++
Console.WriteLine("До сериализации:");
Console.WriteLine(dataXml1);
//Формирование имени файла
string fileXml1Name = Path.Combine(currentPath, "file2.xml");
//При сериализации файл необходимо открыть с использованием потоков
Stream TestFileStream1Xml1 = File.Create(fileXml1Name);
//Создание объекта класса сериализации
XmlSerializer serializerXml1 = new XmlSerializer(typeof(DataXml1));
//Сериализация объекта в файл
serializerXml1.Serialize(TestFileStream1Xml1, dataXml1);
//Закрытие потока
TestFileStream1Xml1.Close();

//+++++
//десериализация
//+++++
//Открытие файла в виде потока на чтение
Stream TestFileStream2Xml1 = File.OpenRead(fileXml1Name);
//Десериализация объекта
DataXml1 dataXml1Out =
(DataXml1)serializerXml1.Deserialize(TestFileStream2Xml1);
//Закрытие потока
TestFileStream2Xml1.Close();
Console.WriteLine("После десериализации:");
Console.WriteLine(dataXml1Out);

```

Рассмотренный пример практически полностью совпадает с бинарной сериализацией/десериализацией, за исключением использования класса XmlSerializer. Конструктор класса XmlSerializer требует передачи в качестве параметра информации о типе сериализуемого класса «typeof(DataXml1)».

Результаты вывода в консоль:

Сериализация/десериализация в формате XML:

До сериализации:

str1=строка1 xml int1=3333 double1=333,33

После десериализации:

str1=строка1 xml int1=3333 double1=333,33

Отличие от бинарной сериализации состоит в том, что при сериализации формируется не бинарный файл, а файл в формате XML. Содержимое файла «file2.xml»:

```
<?xml version="1.0"?>
<DataXml1
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <int1>3333</int1>
  <double1>333.33</double1>
  <str1>строка1 xml</str1>
</DataXml1>
```

При использовании XML-сериализации существует возможность с помощью аннотаций для класса и свойств задавать формат сериализации. Изменим класс данных, добавив аннотации для сериализации:

```
using System;
using System.Xml.Serialization;

namespace Serialization
{
    [XmlRoot(ElementName = "DataXmlWithParameters")]
    public class DataXml2
    {
        [XmlIgnore]
        public int int1 { get; set; }
        [XmlAttribute(AttributeName = "DoubleParameter")]
        public double double1 { get; set; }
        [XmlElement(ElementName = "StringParameter")]
        public string str1 { get; set; }

        public override string ToString()
        {
            return
                "str1=" + str1
                + " int1=" + int1.ToString()
                + " double1=" + double1.ToString();
        }
    }
}
```

В данном примере класс помечен аннотацией XmlRoot, в качестве параметра указано имя элемента XML «DataXmlWithParameters». Свойство int1 помечено аннотацией XmlIgnore, что отменяет сериализацию данного свойства. Свойство double1 помечено аннотацией XmlAttribute, поэтому



оно будет сериализовано в виде XML-атрибута с именем «DoubleParameter», а свойство str1 будет сериализовано в виде XML-элемента с именем «StringParameter».

Содержимое XML-файла для класса DataXml2:

```
<?xml version="1.0"?>
<DataXmlWithParameters
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  DoubleParameter="333.33">
  <StringParameter>строка1 xml</StringParameter>
</DataXmlWithParameters>
```

Имена всех XML-элементов соответствуют именам, указанным в аннотациях класса DataXml2. В соответствии с аннотацией, свойство double1 сериализовано в виде атрибута DoubleParameter.

Создание объекта класса данных, сериализация/десериализация практически полностью аналогичны предыдущему примеру:

```
Console.WriteLine("Сериализация/десериализация в формате XML с
атрибутами:");

DataXml2 dataXml2 = new DataXml2
{
    str1 = "строка1 xml",
    int1 = 3333,
    double1 = 333.33
};

//+++++
//сериализация
//+++++
Console.WriteLine("До сериализации:");
Console.WriteLine(dataXml2);
//Формирование имени файла
string fileXml2Name = Path.Combine(currentPath, "file2atr.xml");
//При сериализации файл необходимо открыть с использованием потоков
Stream TestFileStream1Xml2 = File.Create(fileXml2Name);
//Создание объекта класса сериализации
XmlSerializer serializerXml2 = new XmlSerializer(typeof(DataXml2));
//Сериализация объекта в файл
serializerXml2.Serialize(TestFileStream1Xml2, dataXml2);
//Закрытие потока
TestFileStream1Xml2.Close();

//+++++
```

```

//десериализация
//+++++
//Открытие файла в виде потока на чтение
Stream TestFileStream2Xml2 = File.OpenRead(fileXml2Name);
//Десериализация объекта
DataXml2 dataXml2Out =
(DataXml2)serializerXml2.Deserialize(TestFileStream2Xml2);
//Закрытие потока
TestFileStream2Xml2.Close();
Console.WriteLine("После десериализации:");
Console.WriteLine(dataXml2Out);

```

Результаты вывода в консоль:

Сериализация/десериализация в формате XML с атрибутами:

До сериализации:

```
str1=строка1 xml int1=3333 double1=333,33
```

После десериализации:

```
str1=строка1 xml int1=0 double1=333,33
```

Как и ожидалось, после десериализации свойство `int1` не восстановлено (имеет значение по умолчанию 0), так как данное свойство помечено атрибутом `XmlIgnore`.

## 8 Рефлексия

В данном разделе рассматриваются основы механизма рефлексии. Рефлексия с одной стороны включает достаточно большое количество сведений, авторы [1] отмечают, что рефлексии можно посвятить отдельную книгу. С другой стороны, рефлексия очень широко используется в современном C#. Как ни старались авторы данного пособия излагать материал последовательно, в предыдущих разделах пособия рефлексия использовалась в большом количестве примеров:

- для определения типа данных при работе с необобщенными коллекциями (работа с типами данных);
- для сортировки коллекций используется проверка, что тип реализует интерфейс `Comparable` (работа с типами данных);

- для определения пути к исполняемому файлу (работа со сборками);
- для бинарной сериализации класс должен быть помечен атрибутом «[Serializable]» (работа с атрибутами).

Фактически работа со сборками, типами данных и атрибутами являются основными задачами рефлексии.

Также хочется отметить, что в некоторых книгах термин «рефлексия» («reflection») иногда переводят с английского как «отражение». Но с таким переводом трудно согласиться. Дело в том, что данный термин соответствует философскому термину «рефлексия». В соответствии с философским энциклопедическим словарем «рефлексия (от лат. Reflexio – обращение назад) – способность человеческого мышления к критическому самоанализу». Если спроецировать это определение на язык информатики, то можно определить рефлексию как способность языка программирования анализировать собственные программы различными способами.

Рассмотрим более детально возможности рефлексии в .NET на основе **примера 14**.

### **8.1 Работа со сборками**

Классы для работы с рефлексией расположены в пространстве имен System.Reflection.

Класс Assembly предназначен для получения информации о сборках. Метод GetExecutingAssembly возвращает информацию о текущей исполняемой сборке. Также существует группа методов Load\* (Load, LoadFile, LoadFrom), которые позволяют загрузить сборку и получить информацию о ней.

Пример кода, выводящего в консоль информацию о текущей сборке:

```
Console.WriteLine("Вывод информации о сборке:");  
Assembly i = Assembly.GetExecutingAssembly();
```

```
Console.WriteLine("Полное имя:" + i.FullName);
Console.WriteLine("Исполняемый файл:" + i.Location);
```

Результаты вывода в консоль:

Вывод информации о сборке:

```
Полное имя: Reflection, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Исполняемый файл: C:\root\13\Reflection\Reflection\bin\Debug\Reflection.exe
```

## 8.2 Работа с типами данных

Работа с типами данных один из наиболее часто применяемых механизмов рефлексии.

Создадим класс ForInspection, который содержит информацию для получения информации с помощью рефлексии.

```
using System;

namespace Reflection
{
    /// <summary>
    /// Класс для исследования с помощью рефлексии
    /// </summary>
    public class ForInspection : IComparable
    {
        public ForInspection() { }
        public ForInspection(int i) { }
        public ForInspection(string str) { }

        public int Plus(int x, int y) { return x + y; }
        public int Minus(int x, int y) { return x - y; }

        [NewAttribute("Описание для property1")]
        public string property1
        {
            get { return _property1; }
            set { _property1 = value; }
        }
        private string _property1;

        public int property2 { get; set; }

        [NewAttribute(Description = "Описание для property3")]
        public double property3 { get; private set; }

        public int field1;
        public float field2;
```

```

    /// <summary>
    /// Реализация интерфейса IComparable
    /// </summary>
    public int CompareTo(object obj)
    {
        return 0;
    }
}
}

```

Класс реализует интерфейс IComparable, содержит конструкторы, методы, свойства, поля данных. Для получения информации о типе (классе) ForInspection необходимо создать объект класса Type.

Отметим, что строго говоря, рефлексия не является объектно-ориентрованным механизмом. Для получения информации о типах разработчики C# могли разработать отдельный языковой механизм. Но поскольку C# является объектно-ориентрованным языком, то и рефлексии типов разработчики C# реализовали на основе объектно-ориентрованного подхода. Type – это специализированный класс, содержащий информацию о других классах. Объект класса Type содержит информацию о конкретном классе.

Получить информацию о типе можно двумя способами. Если создан объект класса, то получить информацию о классе можно с помощью метода GetType. Метод GetType унаследован от класса object, поэтому он должен присутствовать у любого объекта C#. Пример:

```

ForInspection obj = new ForInspection();
Type t = obj.GetType();

```

Если объект класса не создан, то можно использовать оператор typeof.

Пример:

```

Type t = typeof(ForInspection);

```

Пример получения информации о типе ForInspection с использованием класса Type:

```

ForInspection obj = new ForInspection();
Type t = obj.GetType();
//другой способ
//Type t = typeof(ForInspection);

```

```

Console.WriteLine("\nИнформация о типе:");
Console.WriteLine("Тип " + t.FullName + " унаследован от " +
t.BaseType.FullName);
Console.WriteLine("Пространство имен " + t.Namespace);
Console.WriteLine("Находится в сборке " + t.AssemblyQualifiedName);

Console.WriteLine("\nКонструкторы:");
foreach (var x in t.GetConstructors())
{
    Console.WriteLine(x);
}

Console.WriteLine("\nМетоды:");
foreach (var x in t.GetMethods())
{
    Console.WriteLine(x);
}

Console.WriteLine("\nСвойства:");
foreach (var x in t.GetProperties())
{
    Console.WriteLine(x);
}

Console.WriteLine("\nПоля данных (public):");
foreach (var x in t.GetFields())
{
    Console.WriteLine(x);
}

Console.WriteLine("\nForInspection реализует IComparable -> " +
t.GetInterfaces().Contains(typeof(IComparable))
);

```

Таким образом, класс `Type` содержит свойства и методы, которые позволяют получить информацию о конструкторах, методах, свойствах, полях данных исследуемого класса, проверить от какого класса наследуется класс, какие реализует интерфейсы.

Результаты вывода в консоль:

```

Информация о типе:
Тип Reflection.ForInspection унаследован от System.Object
Пространство имен Reflection
Находится в сборке Reflection.ForInspection, Reflection, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null

```

Конструкторы:

Void .ctor()

Void .ctor(Int32)

Void .ctor(System.String)

Методы:

Int32 Plus(Int32, Int32)

Int32 Minus(Int32, Int32)

System.String get\_property1()

Void set\_property1(System.String)

Int32 get\_property2()

Void set\_property2(Int32)

Double get\_property3()

Int32 CompareTo(System.Object)

System.String ToString()

Boolean Equals(System.Object)

Int32 GetHashCode()

System.Type GetType()

Свойства:

System.String property1

Int32 property2

Double property3

Поля данных (public):

Int32 field1

Single field2

ForInspection реализует IComparable -> True

### **8.3 Динамические действия с объектами классов**

Метод `InvokeMember` класса `Type` позволяет осуществлять динамические действия с объектами классов: создавать объекты, вызывать методы, получать и присваивать значения свойств и др.

Особенность заключается в том, что имена свойств, классов передаются методу `InvokeMember` в виде строковых параметров.

В следующем примере с использованием метода `InvokeMember` создается объект класса `ForInspection` и вызывается его метод:

```
Type t = typeof(ForInspection);
Console.WriteLine("\nВызов метода:");
//Создание объекта
//ForInspection fi = new ForInspection();
//Можно создать объект через рефлексию
ForInspection fi =
    (ForInspection)t.InvokeMember(
        null, BindingFlags.CreateInstance,
        null, null, new object[] { });
//Параметры вызова метода
object[] parameters = new object[] { 3, 2 };
//Вызов метода
object Result =
    t.InvokeMember("Plus", BindingFlags.InvokeMethod,
        null, fi, parameters);
Console.WriteLine("Plus(3,2)={0}", Result);
```

Метод `InvokeMember` принимает различные параметры: имя метода или свойства, к которому происходит обращение, список аргументов вызываемого метода и др.

Одним из важнейших параметров является параметр типа `BindingFlags`. `BindingFlags` – это перечисление, которое определяет выполняемое действие: создание объекта, обращение к методу или свойству и т.д.

Результаты вывода в консоль:

```
Вызов метода:
Plus(3,2)=5
```

## 8.4 Работа с атрибутами

Работа с атрибутами, как и работа с типами данных является одним из наиболее часто применяемых механизмов рефлексии.

Поскольку в `C#` рефлексия реализована с использованием объектно-ориентированного подхода, то атрибуты являются классами. Создадим класс атрибута:

```
using System;
```



```

namespace Reflection
{
    /// <summary>
    /// Класс атрибута
    /// </summary>
    [AttributeUsage(AttributeTargets.Property, AllowMultiple =
false, Inherited = false)]
    public class NewAttribute : Attribute
    {
        public NewAttribute() { }
        public NewAttribute(string DescriptionParam)
        {
            Description = DescriptionParam;
        }

        public string Description { get; set; }
    }
}

```

Если класс используется в качестве атрибута, то он должен быть унаследован от класса «System.Attribute». Класс может содержать любые данные и методы, в данном случае у класса есть два конструктора (с параметром и без параметра) и автоопределяемое свойство Description.

Также класс атрибута должен быть в свою очередь помечен атрибутом «AttributeUsage», который принимает три параметра:

- параметр типа перечисление AttributeTargets, которое указывает, к каким элементам класса может применяться атрибут NewAttribute (классам, свойствам, методам и т.д.), в данном случае только к свойствам (Property);
- логический параметр AllowMultiple, который указывает, может ли применяться к свойству несколько атрибутов NewAttribute, в данном случае это запрещено;
- логический параметр Inherited, который указывает, наследуется ли атрибут классами, производными от класса с атрибутами, как правило, используется значение false.

Применение нескольких атрибутов может использоваться, например, в тех случаях, когда с помощью атрибутов помечают сделанные

изменения, каждое изменение может помечаться атрибутом с указанием даты-времени и описания изменения.

Использование атрибута `NewAttribute` рассмотрим на примере класса `ForInspection`. Фрагмент класса, который содержит использование атрибутов:

```
[NewAttribute("Описание для property1")]
public string property1
{
    get { return _property1; }
    set { _property1 = value; }
}
private string _property1;

[NewAttribute(Description = "Описание для property3")]
public double property3 { get; private set; }
```

Атрибут необходимо записывать в квадратных скобках непосредственно перед тем элементом класса, к которому он относится, в данном случае атрибут применяется к свойствам.

Каждая запись в квадратных скобках является объектом соответствующего класса атрибута, в данном случае объектом класса `NewAttribute`. В данном примере мы «приклеиваем стикер» в виде объекта класса `NewAttribute` к соответствующему свойству класса `ForInspection`, аннотируем свойства некоторой дополнительной информацией.

При создании аннотации механизм `IntelliSense` автоматически определяет конструкторы класса-атрибута. Все `public`-свойства автоматически превращаются в именованные параметры. Пример использования `IntelliSense` для аннотации показан на рис. 22.

Отметим, что информация на рис. 22 полностью соответствует определению класса `NewAttribute`. Действительно, класс `NewAttribute` содержит два конструктора (пустой и с одним строковым параметром), а также `public`-свойство `Description`, которое `IntelliSense` определяет как именованный параметр аннотации.

```

[NewAttribute(
  ▲ 1 из 4 ▼ NewAttribute.NewAttribute() get; private set; }
  ▲ 2 из 4 ▼ NewAttribute.NewAttribute(Именованные параметры...)
  Именованные параметры: Description = string
  ▲ 3 из 4 ▼ NewAttribute.NewAttribute(string DescriptionParam)
  ▲ 4 из 4 ▼ NewAttribute.NewAttribute(string DescriptionParam, Именованные параметры...)
  /// Реализация интерфейса IComparable
  /// </summary>
  public int CompareTo(object obj)
  {
    return 0;
  }

```

Рис. 22. Работа IntelliSense для аннотаций.

При компиляции аннотации также компилируются и записываются в соответствующую сборку. Далее с помощью механизма рефлексии можно проверить снабжен ли элемент класса какой-либо аннотацией. В нашем примере такая проверка вынесена в отдельную функцию:

```

/// <summary>
/// Проверка, что у свойства есть атрибут заданного типа
/// </summary>
/// <returns>Значение атрибута</returns>
public static bool GetPropertyAttribute(PropertyInfo checkType, Type
attributeType, out object attribute)
{
  bool Result = false;
  attribute = null;

  //Поиск атрибутов с заданным типом
  var isAttribute =
    checkType.GetCustomAttributes(attributeType, false);
  if (isAttribute.Length > 0)
  {
    Result = true;
    attribute = isAttribute[0];
  }

  return Result;
}

```

Метод принимает в качестве параметров информацию о проверяемом свойстве «PropertyInfo checkType» и тип проверяемого атрибута «Type

attributeType». Если проверяемое свойство содержит атрибуты данного типа (метод «checkType.GetCustomAttributes» возвращает коллекцию isAttribute из более чем одного элемента), то метод возвращает истину, а в выходной параметр метода «out object attribute» помещается первый элемент коллекции isAttribute (в соответствии с определением свойство NewAttribute может применяться к свойству не более одного раза).

Пример, осуществляющий проверку того, что свойства снабжены атрибутом NewAttribute. Если атрибут используется, то выводится содержимое поля Description:

```
Type t = typeof(ForInspection);
Console.WriteLine("\nСвойства, помеченные атрибутом:");
foreach (var x in t.GetProperties())
{
    object attrObj;
    if (GetPropertyAttribute(x, typeof(NewAttribute), out attrObj))
    {
        NewAttribute attr = attrObj as NewAttribute;
        Console.WriteLine(x.Name + " - " + attr.Description);
    }
}
```

В данном примере в цикле перебираются все свойства класса ForInspection. Если свойство снабжено атрибутом, то выводится название свойства и поле Description атрибута.

Информация о свойстве получается с помощью рассмотренной функции GetPropertyAttribute. Для приведения полученного значения типа object к требуемому типу NewAttribute используется оператор as в форме «выражение as тип». Оператор as является другой формой приведения типов кроме рассмотренного ранее приведения типов с помощью оператора «круглые скобки» в форме «(тип)выражение».

Результаты вывода в консоль:

```
Свойства, помеченные атрибутом:
property1 - Описание для property1
property3 - Описание для property3
```

Атрибуты очень широко применяются в .NET. Например, в технологии ASP.NET MVC с помощью атрибутов задаются правила проверки полей ввода при заполнении форм данных.

Отметим, что в Java существует аналогичный атрибутам механизм, который называется аннотациями. Вместо квадратных скобок для выделения аннотаций используется символ «@».

### **8.5 Использование рефлексии на уровне откомпилированных инструкций**

В начальных разделах пособия говорилось о том, что .NET является программно-реализованной моделью микропроцессора. Поэтому язык машинных команд MSIL (IL) можно представить в виде команд специфического диалекта языка ассемблер.

Для дизассемблирования необходимо запустить утилиту «ildasm» - IL Disassembler. Данная утилита входит в комплект Visual Studio. Для запуска необходимо открыть «командную строку разработчика Visual Studio» (данный пункт присутствует в меню «пуск» в разделе Visual Studio) и набрать в командной строке команду «ildasm».

Данная утилита является оконной, хотя содержит ключи для запуска в командной строке. При запуске без ключей в командной строке открывается окно, в котором необходимо выбрать пункт меню «Файл/Открыть» и выбрать соответствующий файл сборки. Выберем в качестве примера файл «Reflection.exe» из рассматриваемого в данном разделе **примера 14**.

В результате открывается окно содержащее список классов сборки. Если раскрыть соответствующий класс (в нашем примере ForInspection), то отобразится список конструкторов, методов, свойств, что показано на рис. 23.

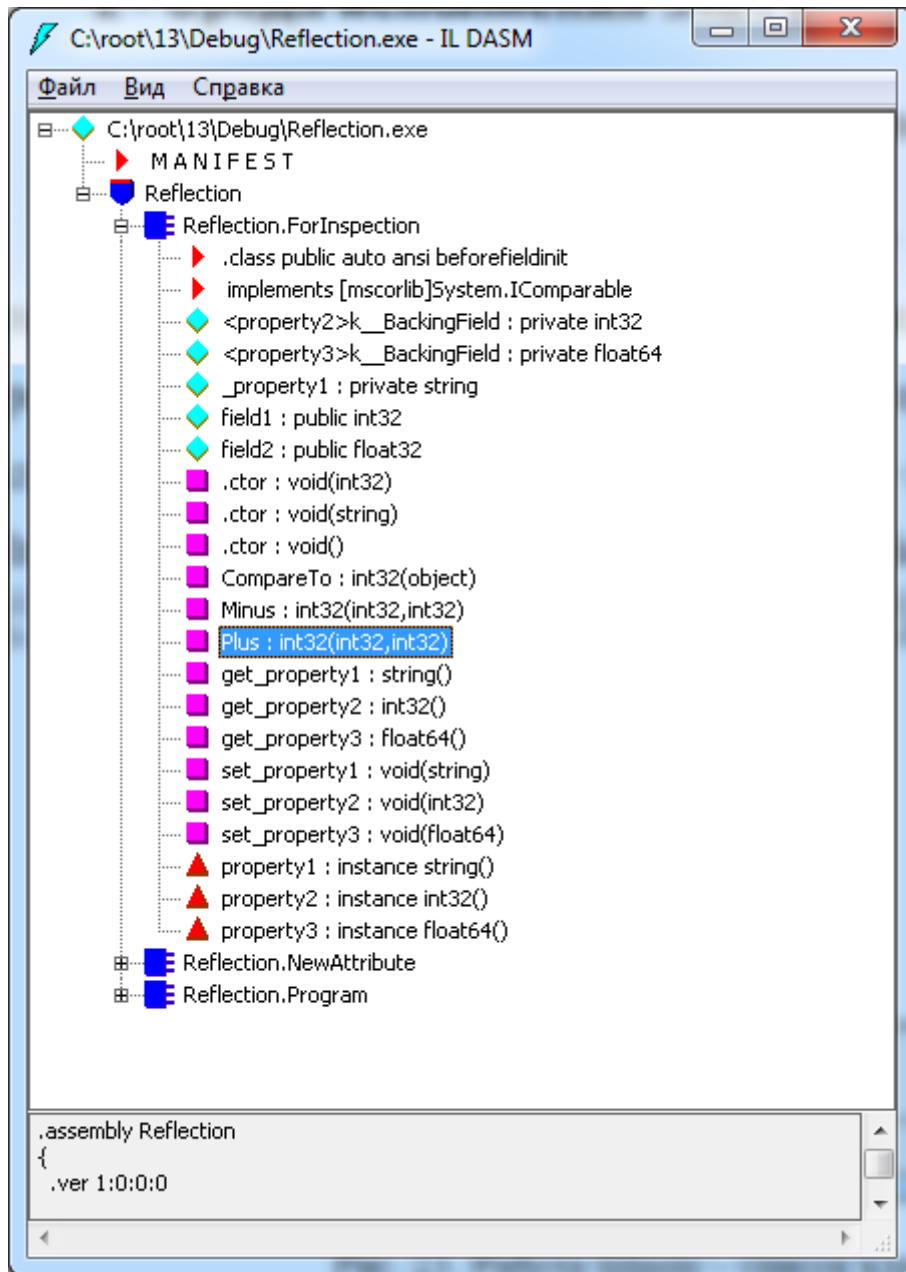


Рис. 23. Работа ildasm – список классов сборки.

Если дважды кликнуть на определение любого элемента, то детальные команды языка IL отобразятся в отдельном окне. Нажмем дважды на определение метода Plus, что показано на рис. 24.

```

Reflection.ForInspection::Plus : int32(int32,int32)
Найти Найти далее
.method public hidebysig instance int32 Plus(int32 x,
                                             int32 y) cil managed
{
    // Размер кода:          9 (0x9)
    .maxstack 2
    .locals init ([0] int32 CS$1$0000)
    IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldarg.2
    IL_0003: add
    IL_0004: stloc.0
    IL_0005: br.s          IL_0007
    IL_0007: ldloc.0
    IL_0008: ret
} // end of method ForInspection::Plus

```

Рис. 24. Работа ildasm – код метода.

Метод Plus осуществляет сложение двух целых чисел. Код на C#:

```
public int Plus(int x, int y) { return x + y; }
```

Можно отметить некоторое сходство команд языка IL и команд ассемблера для Intel x86. Команда add осуществляет сложение двух элементов стека виртуальной машины, ldarg загружает параметр метода в стек, ret осуществляет выход из метода.

Также отметим, что .NET предоставляет набор классов в пространстве имен System.Reflection.Emit, которые предназначены для динамической генерации сборок. Эти классы могут быть полезны для разработчиков новых языков на платформе .NET а также для более детального знакомства с MSIL. В данное пространство имен входят классы для создания сборок, классов, методов и др. В частности, класс ILGenerator позволяет генерировать бинарный код MSIL. Класс OpCodes содержит полный список всех команд MSIL.

Существуют программы-декомпиляторы, которые пытаются по коду MSIL восстановить исходный код на C#, примером такого проекта является .NET Reflector.

Но такому восстановлению могут препятствовать программы-обфускаторы, которые осуществляют обфускацию (запутывание) откомпилированного кода. Если сборка подвергнута обфускации, то она работает корректно. Но команды MSIL расположены в сборке таким образом, что декомпилятор не может восстановить по ним более высокоуровневые команды на C#. Примером такого проекта является Obfuscator.

## **8.6 Самоотображаемость**

Самоотображаемость (англ. homoiconicity) – это способность языка программирования анализировать программу на этом языке как структуру данных.

Термин «самоотображаемость» часто используют вместе с термином «метапрограммирование». Самоотображаемость предполагает анализ программ как структур данных, а метапрограммирование предполагает генерацию на основе этих структур данных других программ.

Можно сказать, что самоотображаемость является предельным случаем рефлексии, так как в этом случае нет никаких ограничений для анализа. Нет необходимости помечать какие-то фрагменты программы атрибутами, читать список методов класса и т.д. – вся программа является структурой данных, в которой можно производить любой поиск и с которой можно выполнять любые действия.

В том числе самоотображаемость предполагает работу с данными на уровне операторов языка, то есть на самом детальном уровне.

В настоящее время C# не поддерживает самоотображаемость. Проблема состоит как раз в детальном уровне. C# является компилируемым языком, с использованием рефлексии можно получить откомпилированные команды (как это делалось в предыдущем разделе), но по ним невозможно «на лету» восстановить исходные команды C#.



Изначально самоотображаемость заявлялась разработчиками С# в числе основных приоритетов. Однако в настоящее время разработчики заявляют, что самоотображаемость планируется реализовать в отдаленной перспективе.

Необходимо отметить, что самоотображаемость может быть реализована «поверх» языка с использованием библиотек для работы с AST (Abstract Syntax Tree – абстрактное синтаксическое дерево). AST позволяет представить текст программы в виде дерева, в котором внутренние вершины соответствуют операторам языка программирования или функциям, а листья соответствуют операндам. Таким образом, AST позволяет представить текст программы в виде структуры данных (дерева) и фактически реализует принцип самоотображаемости. В отличие от получения команд из откомпилированной сборки с использованием рефлексии, AST-дерево строится по исходному тексту программы. В .NET работа с AST-деревом для прикладных программистов реализуется в рамках проекта модульного компилятора Roslyn.

В настоящее время наиболее известным языком, который поддерживает самоотображаемость, является язык Lisp, у которого существует большое количество диалектов, и который исторически является одним из первых языков программирования. В Lisp программа представляется в виде иерархического списка (программа фактически является AST), программа может быть обработана другой программой. На платформе .NET реализован один из современных диалектов Lisp – clojure.

В числе самоотображаемых языков можно также назвать Elixir (синтаксис которого похож на Ruby, но при этом реализована самоотображаемость в стиле Lisp), Prolog (специализированный язык логического программирования).

Самоотображаемость на уровне команд языка ближе к интерпретируемым языкам, которые интерпретируются и выполняются на

уровне исходного кода. Изначально Lisp был интерпретируемым языком. Самоотображаемость для компилируемого языка требует разработки сложного специализированного компилятора. В настоящее время проводятся активные работы по разработке самоотображаемых языков на основе интерпретируемого языка JavaScript.

## 9 Параллельная обработка данных

В любом языке программирования параллельное программирование традиционно считается наиболее сложной темой.

При параллельной обработке могут возникать сложные и плохо отлаживаемые ошибки, связанные с одновременной записью в один и тот же набор данных, одновременным обращением к ресурсам и др. Такие сложные случаи в пособии не рассматриваются.

В данном разделе рассматривается наиболее простой вариант параллельной обработки данных, когда в однородном массиве данных необходимо осуществить поиск. В этом случае возможно разделить массив данных на несколько фрагментов и осуществлять поиск параллельно.

Параллельная обработка данных рассматривается в данном разделе на основе **примера 15**.

В C# существует три вида классов для параллельного запуска потоков:

- Thread (поток);
- ThreadPool (пул потоков);
- Task (задача).

Класс Thread является исторически первым средством для многопоточной работы в C#. Данный класс является практически полной копией такого же класса в языке Java. Основной проблемой класса Thread в C# является медленный запуск потока, на создание и запуск потока уходит довольно много времени.

Для решения этой проблемы был разработан класс `ThreadPool`, который содержит пул уже запущенных потоков, потоки при запуске начинают выполняться без задержки. Проблемой класса `ThreadPool` является не очень удобный для прикладного программиста набор методов для работы с потоками.

Для преодоления этих неудобств был разработан класс `Task`. В настоящее время этот класс является рекомендуемым средством для работы с потоками. Класс `Task` основан на классе `ThreadPool` и обладает всеми его достоинствами, и при этом содержит удобный для прикладного программиста набор методов для работы с потоками.

В данном разделе мы рассмотрим примеры работы с классами `Thread` и `Task` (класс `ThreadPool` не рассматривается), а также пример параллельного поиска в массиве данных на основе класса `Task`.

## 9.1 Использование класса `Thread`

Пример использования класса `Thread`:

```

/// <summary>
/// Пример запуска потоков
/// </summary>
public static void ThreadExample()
{
    Console.WriteLine("Пример использования \"старого\" варианта
многопоточности с использованием класса Thread:");
    const int n = 10;
    Thread[] threads = new Thread[n];
    for (int i = 0; i < n; i++)
    {
        threads[i] = new Thread(ThreadRun);
    }
    for (int i = 0; i < n; i++)
    {
        threads[i].Start("Поток №" + i.ToString());
    }
}

```

В данном примере создается массив из 10 элементов `threads` объектов класса `Thread`. В цикле производится инициализация элементов массива, для этого создаются объекты класса `Thread`.

Конструктор класса `Thread` принимает параметр делегатного типа, который соответствует делегату `ParameterizedThreadStart`:

```
public delegate void ParameterizedThreadStart(object obj);
```

То есть, в конструктор класса `Thread` нужно передать метод, который принимает один параметр типа `object` и возвращает `void`. В данном примере таким методом является `ThreadRun`.

Далее в цикле производится запуск потоков. Для этого вызывается метод `Start` для каждого элемента массива потоков. В качестве параметра в метод `Start` передается объект класса `object`, который подставляется при запуске потока в делегатный параметр `ParameterizedThreadStart`, в нашем примере строка «"Поток №" + `i.ToString()`» будет передана в качестве параметра методу `ThreadRun`.

Рассмотрим более детально метод `ThreadRun`, который вызывается в потоке:

```
/// <summary>
/// Метод, запускаемый в потоке
/// </summary>
/// <param name="param"></param>
public static void ThreadRun(object param)
{
    Random r = new Random();
    int delay = r.Next(3000, 10000);
    Thread.Sleep(delay);
    Console.WriteLine(param.ToString());
}
```

В методе создается объект «`r`» класса `Random`, который используется для генерации случайных чисел. Метод «`r.Next`» возвращает случайное число в диапазоне от 3000 до 10000. Это задержка в миллисекундах (от 3 до 10 секунд).

Полученное случайное время задержки передается в качестве параметра статическому методу `Thread.Sleep`, который осуществляет задержку текущего потока на заданное количество миллисекунд.

После задержки в консоль выводится сообщение, переданное в метод `ThreadRun` в качестве параметра.

Таким образом, параллельно запускается 10 экземпляров метода `ThreadRun`, каждый из которых через случайное время в диапазоне от 3 до 10 секунд выводит в консоль сообщение, которое передано в метод в качестве параметра при инициализации потока.

Результаты вывода в консоль (после завершения работы всех потоков):

Пример использования "старого" варианта многопоточности с использованием класса `Thread`:

```
Поток №7  
Поток №8  
Поток №9  
Поток №5  
Поток №6  
Поток №1  
Поток №0  
Поток №2  
Поток №3  
Поток №4
```

Сообщения выводятся в консоль в случайном порядке, что связано со случайным временем задержки в каждом потоке.

Отметим, что хотя класс `Thread` вполне решает задачу параллельного запуска потоков, его использование связано с определенными неудобствами для прикладного программиста. Все параметры в метод потока передаются через единственный параметр типа `object`. Поэтому, как правило, необходимо создавать отдельный класс, который будет содержать как входные, так и выходные параметры потока, передавать объект данного класса в качестве параметра в метод потока, в методе потока осуществлять приведение типа к требуемому классу, и возвращать через данный класс результаты работы потока, так как класс `Thread` не имеет специальных способов возврата результатов работы в вызывающий поток.

Также отметим, что аналогичные сложности возникают при использовании класса ThreadPool.

## 9.2 Использование класса Task

Рассмотренные выше сложности были учтены разработчиками .NET при проектировании класса Task.

Реализуем рассмотренный ранее пример с использованием класса Task:

```

/// <summary>
/// Пример запуска потоков с использованием Task
/// </summary>
public static void TaskExample()
{
    Console.WriteLine("Пример использования \"нового\" варианта
многопоточности с использованием класса Task:");
    const int n = 10;
    for (int i = 0; i < n; i++)
    {
        Task.Factory.StartNew(
            (object param) =>
            {
                Random r = new Random();
                int delay = r.Next(3000, 10000);
                Thread.Sleep(delay);
                Console.WriteLine(param.ToString());
            },
            "Поток №" + i.ToString());
    }
}

```

В данном примере нет необходимости создавать массив. Класс Task имеет статическое свойство Factory (фабрика). Здесь предполагается, что это «фабрика» для создания объектов (используется шаблон объектно-ориентированного проектирования «фабрика»). У фабрики есть метод создания и запуска потока StartNew. В данном примере используется следующая перегрузка метода StartNew:

```
public Task StartNew(Action<object> action, object state);
```

Первым параметром метода StartNew является метод, соответствующий делегату «Action<object>», то есть, как и в случае

использования класса Thread метод принимающий параметр типа object и возвращающий void.

В рассматриваемом примере первому параметру соответствует лямбда-выражение, которое является аналогом метода ThreadRun при использовании класса Thread:

```
(object param) =>
{
    Random r = new Random();
    int delay = r.Next(3000, 10000);
    Thread.Sleep(delay);
    Console.WriteLine(param.ToString());
}
```

Вторым параметром метода StartNew является параметр типа object, который подставляется при запуске потока в делегатный параметр.

В рассматриваемом примере второму параметру соответствует строка «"Поток №" + i.ToString()», которая при запуске потока подставляется в параметр лямбда-выражения «(object param) => ».

Результаты вывода в консоль (после завершения работы всех потоков):

Пример использования "нового" варианта многопоточности с использованием класса Task:

```
Поток №0
Поток №5
Поток №1
Поток №3
Поток №2
Поток №7
Поток №6
Поток №4
Поток №9
Поток №8
```

Как и в предыдущем случае сообщения выводятся в консоль в случайном порядке.

Код решения задачи с использованием класса Task получился более компактным, метод потока задается с использованием лямбда-выражения.

Однако более важным преимуществом класса `Task` является возможность возврата результата выполнения потока.

### **9.3 Возврат результата выполнения потока с использованием класса `Task`**

Решим более сложную задачу с использованием класса `Task`. Заполним массив числами от 1 до 100 и найдем в этом массиве числа, которые без остатка делятся на 3. При этом массив разделим на 10 подмассивов (по соответствующим десяткам) и будем выполнять поиск многопоточно. Результаты поиска по 10 подмассивам объединим в единый массив результатов.

Для решения данной задачи необходимо решить подзадачу не связанную с потоками – деление массива на подмассивы. Для решения данной задачи создадим вспомогательный класс для хранения диапазонов:

```

/// <summary>
/// Хранение минимального и максимального значений диапазона
/// </summary>
public class MinMax
{
    public int Min {get; set;}
    public int Max {get; set;}

    public MinMax(int pmin, int pmax)
    {
        this.Min = pmin;
        this.Max = pmax;
    }
}

```

Деление массива на подмассивы реализуется с использованием метода `DivideSubArrays` статического класса `SubArrays`. Код приведен ниже и детально откомментирован:

```

/// <summary>
/// Класс для деления массива на последовательности
/// </summary>
public static class SubArrays
{
    /// <summary>

```



```

/// Деление массива на последовательности
/// </summary>
/// <param name="beginIndex">Начальный индекс массива</param>
/// <param name="endIndex">Конечный индекс массива</param>
/// <param name="subArraysCount">Требуемое количество
подмассивов</param>
/// <returns>Список пар с индексами подмассивов</returns>
public static List<MinMax> DivideSubArrays(
    int beginIndex, int endIndex, int subArraysCount)
{
    //Результирующий список пар с индексами подмассивов
    List<MinMax> result = new List<MinMax>();

    //Если число элементов в массиве слишком мало для деления
    //то возвращается массив целиком
    if ((endIndex - beginIndex) <= subArraysCount)
    {
        result.Add(new MinMax(0, (endIndex - beginIndex)));
    }
    else
    {
        //Размер подмассива
        int delta = (endIndex - beginIndex) / subArraysCount;
        //Начало отсчета
        int currentBegin = beginIndex;
        //Пока размер подмассива укладывается в оставшуюся
        //последовательность
        while ((endIndex - currentBegin) >= 2 * delta)
        {
            //Формируем подмассив на основе начала
            //последовательности
            result.Add(
                new MinMax(currentBegin, currentBegin + delta));
            //Сдвигаем начало последовательности
            //вперед на размер подмассива
            currentBegin += delta;
        }
        //Оставшийся фрагмент массива
        result.Add(new MinMax(currentBegin, endIndex));
    }
    //Возврат списка результатов
    return result;
}
}

```

В результате работы метод `DivideSubArrays` возвращает список объектов класса `MinMax`, которые хранят начальный и конечный индексы соответствующих диапазонов массива.

На этом мы завершает рассмотрение вспомогательных классов, и переходим к методу `ArrayThreadExample`, который осуществляет многопоточный поиск в массиве:

```

/// <summary>
/// Многопоточный поиск в массиве
/// </summary>
public static void ArrayThreadExample(
    int ArrayLength, int ThreadCount, int Divider)
{
    //Результирующий список чисел
    List<int> Result = new List<int>();

    //Создание и заполнение временного списка данных
    List<int> tempList = new List<int>();
    for (int i = 0; i < ArrayLength; i++) tempList.Add(i + 1);

    //Деление списка на фрагменты
    //для параллельного запуска в потоках
    List<MinMax> arrayDivList =
        SubArrays.DivideSubArrays(0, ArrayLength, ThreadCount);
    int count = arrayDivList.Count;

    //Вывод диапазонов деления исходного массива
    for (int i = 0; i < count; i++)
    {
        //Вывод результатов, найденных в каждом потоке
        Console.WriteLine("Диапазон " + i.ToString() + ": " +
            arrayDivList[i].Min + " - " +
            arrayDivList[i].Max);
    }
    Console.WriteLine();

    //Создание таймера
    Stopwatch timer = new Stopwatch();
    //Запуск таймера
    timer.Start();

    //Количество потоков соответствует количеству фрагментов массива
    Task<List<int>>[] tasks = new Task<List<int>>[count];

    //Запуск потоков
    for (int i = 0; i < count; i++)
    {
        //Создание временного списка, чтобы потоки
        //не работали параллельно с одной коллекцией
        List<int> tempTaskList =
            tempList.GetRange(arrayDivList[i].Min,
                arrayDivList[i].Max - arrayDivList[i].Min);
    }
}

```

```

tasks[i] = new Task<List<int>>>(
    //Метод, который будет выполняться в потоке
    ArrayThreadPool,
    //Параметры потока передаются в виде кортежа,
    //чтобы не создавать временный класс
    new Tuple<List<int>, int>(tempTaskList, Divider));

//Запуск потока
tasks[i].Start();
}

//Ожидание завершения всех потоков
Task.WaitAll(tasks);

//Остановка таймера
timer.Stop();

//Объединение результатов полученных из разных потоков
for (int i = 0; i < count; i++)
{
    //Вывод результатов, найденных в каждом потоке
    Console.WriteLine("Поток " + i.ToString() + ": ");
    foreach (var x in tasks[i].Result)
        Console.WriteLine(x.ToString() + " ");
    Console.WriteLine();

    //Добавление результатов конкретного потока
    //в общий массив результатов
    Result.AddRange(tasks[i].Result);
}

//Вывод общего массива результатов
Console.WriteLine("\nМассив из {0} элементов обработан {1}
    потоками за {2}. Найдено: {3}", ArrayLength,
    count, timer.Elapsed, Result.Count);
foreach (int i in Result)
    Console.WriteLine(i.ToString().PadRight(5));
Console.WriteLine();
}

```

Метод принимает три параметра:

- int ArrayLength – количество элементов в массиве, в рассматриваемом примере 100;
- int ThreadCount – количество потоков для одновременного поиска, в рассматриваемом примере 10;
- int Divider – искомый делитель, в рассматриваемом примере 3.

В начале работы метода создается переменная `Result` типа `List<int>`, в которую будут сохраняться найденные в потоках числа, которые без остатка делятся на 3.

Далее создается массив `tempList` типа `List<int>` и заполняется числами от 1 до 100.

Далее с использованием метода `SubArrays.DivideSubArrays` осуществляется деление диапазона от 1 до 100 на поддиапазоны по десяткам. Найденные диапазоны записываются в список `arrayDivList`, содержимое списка выводится в консоль в виде «Диапазон 0: 0 - 10».

Для измерения времени выполнения потоков создается объект класса `Stopwatch`. Данный класс, расположенный в пространстве имен `System.Diagnostics` используется для измерения времени выполнения программы. Класс `Stopwatch` является аналогом секундомера, только время измеряется в долях секунд. Метод `Start` запускает таймер, метод `Stop` останавливает таймер, свойство `Elapsed` возвращает измеренное время в виде объекта класса `TimeSpan`. Класс `TimeSpan` предназначен для хранения интервала времени.

Далее создается массив `tasks` объектов класса `Task`. В данном случае используется обобщенная форма класса `Task` – `Task<List<int>>`. `List<int>` – это тип данных, который будет возвращен методом, работающим в потоке класса `Task`.

Далее производится запуск потоков. Перед запуском каждого потока из массива данных `tempList` выделяется фрагмент данных, относящийся к потоку. Класс `List` не является потокобезопасным, однако это означает, что в него нельзя одновременно записывать данные из нескольких потоков. Чтение данных является потокобезопасным действием для любой коллекции, при условии, что в эту коллекцию одновременно не производится запись. В данном примере выделение фрагментов массива не связано с потокобезопасностью, все потоки могли бы читать данные из

общего массива, и это не привело бы неправильной работе программы. Однако обращение к одному массиву из нескольких потоков могло бы повлиять на корректность измерения времени, так как поиск из нескольких потоков в одном массиве мог бы замедлить выполнение программы.

В .NET существуют специальные классы потокобезопасных коллекций, которые расположены в пространстве имен `System.Collections.Concurrent`. Они допускают одновременную запись данных из нескольких потоков, например потокобезопасный аналог списка – `ConcurrentBag`, потокобезопасный словарь – класс `ConcurrentDictionary`.

Отметим, что у класса `List` существует метод `GetRange`, который копирует элементы в заданном диапазоне индексов.

Далее создаются объекты класса `Task`. В данном примере используется не фабрика, а обычный конструктор, который принимает два параметра: исполняемый в потоке метод (в данном примере `ArrayThreadTask`) и параметр типа `object` который будет передан в качестве аргумента исполняемому методу. В данном примере в качестве второго параметра передается кортеж (`Tuple`), который содержит подмассив чисел для поиска в потоке (выделенный для данного потока фрагмент массива) и искомое число (в рассматриваемом примере 3).

Далее производится запуск потока с помощью метода «`tasks[i].Start()`».

После этого необходимо дождаться завершения работы всех потоков, чтобы получить результаты поиска, это делается с помощью вызова метода «`Task.WaitAll`». В качестве параметра метод `WaitAll` принимает массив `tasks`. Метод `WaitAll` завершит работу только после того как отработают все потоки массива `tasks`.

После завершения работы потоков можно получить результаты работы потоков. Для получения результата работы потока используется свойство «`tasks[i].Result`». Свойство `Result` возвращает результат работы метода потока (в данном примере метода `ArrayThreadTask`). Метод

ArrayThreadTask должен возвращать тип данных, указанный в качестве обобщения класса Task, в рассматриваемом примере List<int>. Данные этого типа возвращает свойство Result. Таким образом, Result является хранилищем результата работы метода потока, получить этот результат можно непосредственно через поток (объект класса Task).

Далее производится копирование полученного свойства «tasks[i].Result» для каждого потока в общий список результатов (локальная переменная Result). Отметим, что у класса List существует метод AddRange, который добавляет в список все элементы другого списка.

В консоль выводится информация о числах, найденных в результате поиска, времени поиска и т.д.

Рассмотрим исполняемый в потоке метод ArrayThreadTask:

```

/// <summary>
/// Выполняется в параллельном потоке для поиска числа
/// </summary>
/// <param name="param"></param>
public static List<int> ArrayThreadTask(object paramObj)
{
    //Получение параметров
    Tuple<List<int>, int> param = (Tuple<List<int>, int>)paramObj;
    int listCount = param.Item1.Count;

    //Временный список для результата
    List<int> tempData = new List<int>();

    //Перебор нужных элементов в списке данных
    for (int i = 0; i < listCount; i++)
    {
        //Текущее значение из массива
        int temp = param.Item1[i];

        //Если число делится без остатка на делитель,
        //то сохраняем в результат
        if ((temp % param.Item2) == 0)
        {
            tempData.Add(temp);
        }
    }

    //Возврат массива данных

```

```
    return tempData;  
}
```

В данном методе сначала полученный параметр приводится к типу кортежа, далее из кортежа получается информация о подмассиве для поиска (переменная `param.Item1`) и искомом делителе (переменная `param.Item2`).

Данные подмассива для поиска перебираются в цикле. Если элемент подмассива делится без остатка на искомый делитель, то элемент массива добавляется в результирующий список.

Метод возвращает результирующий список чисел, которые делятся на искомый делитель без остатка. Отметим, что тип возвращаемого значения метода «`List<int>`» совпадает с обобщенным типом, который использовался при объявлении класса `Task`.

В рассматриваемом примере метод `ArrayThreadExample` вызывается следующим образом:

```
ArrayThreadExample(100, 10, 3);
```

Результаты работы метода (вывод в консоль):

```
Диапазон 0: 0 - 10  
Диапазон 1: 10 - 20  
Диапазон 2: 20 - 30  
Диапазон 3: 30 - 40  
Диапазон 4: 40 - 50  
Диапазон 5: 50 - 60  
Диапазон 6: 60 - 70  
Диапазон 7: 70 - 80  
Диапазон 8: 80 - 90  
Диапазон 9: 90 - 100
```

```
Поток 0: 3 6 9  
Поток 1: 12 15 18  
Поток 2: 21 24 27 30  
Поток 3: 33 36 39  
Поток 4: 42 45 48  
Поток 5: 51 54 57 60  
Поток 6: 63 66 69
```

Поток 7: 72 75 78

Поток 8: 81 84 87 90

Поток 9: 93 96 99

Массив из 100 элементов обработан 10 потоками за 00:00:00.0275901. Найдено:  
33

3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48  
51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96  
99

Таким образом, класс Task фактически позволяет вызывать потоки как асинхронные методы, которые возвращают результаты после выполнения.

#### 9.4 Использование конструкций *async* и *await*

В .NET 4.5 для облегчения асинхронной работы с классом Task в язык C# были добавлены новые ключевые слова *async* и *await*.

Рассмотрим использование этих ключевых слов на простейшем примере.

Объявим следующий метод:

```
/// <summary>
/// Задача для запуска
/// </summary>
static Task<string> GetTaskAsync()
{
    //Создание задачи
    Task<string> t = new Task<string>(
        () =>
        {
            Thread.Sleep(3000);
            return "После задержки в 3 секунды.";
        }
    );
    //Запуск задачи
    t.Start();
    //Задача возвращается в качестве результата
    return t;
}
```

Данный метод создает и запускает задачу, которая после задержки в 3 секунды возвращает строковое значение. Несколько необычным является



то, что сама задача (Task), а не результат ее работы, возвращается из метода. Фактически метод возвращает запущенный поток.

Объявим следующий метод с использованием новых ключевых слов:

```

/// <summary>
/// Асинхронный метод
/// </summary>
static async void DelayAsync()
{
    string Result = await GetTaskAsync();
    Console.WriteLine(Result);
}

```

Данный метод объявлен как асинхронный (при объявлении метода используется ключевое слово `async`). Это означает, что в нем могут выполняться длительные действия, данный метод будет запущен асинхронно, не дожидаясь завершения данного метода, будут выполняться следующие после него команды.

В теле метода производится вызов метода `GetTaskAsync`, который возвращает `Task`. При вызове метода `GetTaskAsync` используется ключевое слово `await`, которое ожидает завершения работы объекта класса `Task` и возвращает свойство `Result` объекта класса `Task`. После работы метода `GetTaskAsync`, который возвращает `Task<string>`, команда `await` возвращает значение типа `string`. Полученная строка выводится в консоль.

Вызовем метод `DelayAsync` следующим образом:

```

DelayAsync();
Console.WriteLine("\nПеред вызовом DelayAsync");

```

Поскольку метод `DelayAsync` объявлен как асинхронный, то метод будет запущен и управление будет сразу передано следующей команде – `Console.WriteLine`.

Далее в методе `DelayAsync` будет вызван метод `GetTaskAsync`, который реализует задержку в три секунды и возвращает строковый результат, который будет выведен после задержки.

Таким образом, команда «Console.WriteLine» будет действительно выполнена до завершения асинхронного метода DelayAsync и сообщения в консоль будут выведены в следующем порядке:

Перед вызовом DelayAsync  
После задержки в 3 секунды.

## **10 Реализация алгоритма поиска с опечатками**

Данное пособие предназначено, прежде всего, для изучения языка программирования C# и в нем не ставится цель изучения алгоритмов.

Но в качестве учебной задачи мы рассмотрим один достаточно сложный алгоритм, который довольно часто используется при разработке информационных систем. Это алгоритм поиска в текстовой строке с опечатками.

Практически любая крупная информационная система содержит такие данные как фамилии, имена, отчества, географические названия, которые могут быть набраны с опечатками при вводе данных или при вводе ключевого слова для поиска.

Устранение опечаток при поиске представляет собой нетривиальную задачу. Для решения этой задачи предлагались различные методы, например разбиение слов на два или три символа с их комбинаторным перебором. Но все эти методы оказались мало эффективны.

### ***10.1 Расстояние Дамерау-Левенштейна***

Предложить эффективный способ решения данной задачи удалось отечественному исследователю Владимиру Иосифовичу Левенштейну.

Теоретические выкладки данного раздела основаны на источнике [4].

Вычисление расстояния Левенштейна (редакционного расстояния) основано на понятии редакционного предписания.

Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так:

- D (англ. delete) — удалить,
- I (insert) — добавить,
- R (replace) — заменить,
- M (match) — совпадение.

Пример редакционного предписания представлен на рис. 25:

Действие	<b>M</b>	<b>M</b>	<b>I</b>	<b>R</b>	<b>M</b>	<b>M</b>	<b>R</b>	<b>D</b>
Исходное слово	П	Р		И	М	Е	Р	Ы
Результат	П	Р	Е	Д	М	Е	Т	

Рис. 25. Пример редакционного предписания.

Для преобразования слова «ПРИМЕРЫ» в слово «ПРЕДМЕТ» необходимо выполнить 4 действия (добавление, удаление и 2 замены).

Расстоянием Левенштейна называется минимальное количество действий, необходимых для преобразования одного слова в другое. В этом примере расстояние Левенштейна равно 4.

Преобразовать одно слово в другое можно различными способами, количество действий также может быть разным. Поэтому при вычислении расстояния и выбирается минимальное количество действий.

Если поиск производится в тексте, который набирается на клавиатуре, то вместо расстояния Левенштейна используют усовершенствованное расстояние Дамерау-Левенштейна.

Фредерик Дамерау – один из американских пионеров-исследователей в области обработки текстов на естественном языке.

Исследования Дамерау показали, что наиболее частой ошибкой при наборе слова является перестановка двух соседних букв – транспозиция, обозначается как T (transposition).

Данное действие принято называть поправкой Дамерау к расстоянию Левенштейна.

В случае одной транспозиции расстояние Левенштейна равняется двум (две замены символа). При использовании поправки Дамерау транспозиция считается как единичное расстояние, что показано на рис. 26.

Действие (расстояние Дамерау-Левенштейна)	М	М	Т		М	М	М
Действие (расстояние Левенштейна)	М	М	Р	Р	М	М	М
Исходное слово	П	Р	И	М	Е	Р	Ы
Результат	П	Р	М	И	Е	Р	Ы

Рис. 26. Транспозиция при вычислении расстояний Левенштейна и Дамерау-Левенштейна.

При использовании расстояния Дамерау-Левенштейна за единичное расстояние принимаются следующие действия:

- I (insert) — добавление символа.
- D (delete) — удаление символа.
- R (replace) — замена символа.
- T (transposition) — перестановка двух соседних символов.

## 10.2 Вычисление расстояния Дамерау-Левенштейна

Пусть заданы строка  $S_1$  длиной  $M$  символов и строка  $S_2$  длиной  $N$  символов. При этом предполагается, что строка  $S_1$  длиннее или равна по длине строке  $S_2$  ( $M \geq N$ ).

Для вычисления расстояния Левенштейна необходимо рекуррентно вычислить значения элементов матрицы  $D[i, j]$  размером  $M \times N$ , значение в правом нижнем элементе  $D[M, N]$  и будет являться расстоянием Левенштейна.

Расстояние Левенштейна  $ed(S_1, S_2)$  вычисляется на основе формулы

1:

$ed(S_1, S_2) = D[M, N]$ , где

$$D[i, j] = \begin{cases} 0 & ; i = 0, j = 0 \\ & \text{(случай 1)} \\ i & ; i > 0, j = 0 \\ & \text{(случай 2)} \\ j & ; i = 0, j > 0 \\ & \text{(случай 3)} \\ \min( & \\ & D[i - 1, j] + 1, \text{ (утверждение 1)} \\ & D[i, j - 1] + 1, \text{ (утверждение 2)} \\ & D[i - 1, j - 1] + m(S_1[i], S_2[j]) \text{ (утверждение 3)} \\ ) & ; i > 0, j > 0 \\ & \text{(случай 4)} \end{cases} \quad (1)$$

Выражение  $m(\text{символ1}, \text{символ2})$  равняется нулю, если символы совпадают и единице, если символы не совпадают (проверяется равенство  $i$ -го символа из первой строки и  $j$ -го символа из второй строки).

Рассмотрим формулу более подробно в соответствии с [4].  $D(i, j)$  – расстояние между префиксами строк: первыми  $i$  символами строки  $S_1$  и первыми  $j$  символами строки  $S_2$ .

Редакционное расстояние между двумя пустыми строками равно нулю (случай 1).

Для получения пустой строки из строки длиной  $i$ , нужно совершить  $i$  операций удаления (случай 2).

Для получения строки длиной  $j$  из пустой, нужно произвести  $j$  операций вставки (случай 3).

В случае 4 обе строки непусты. В оптимальной последовательности операций, операции можно произвольно менять местами. Рассмотрим две последовательные операции:

- Две замены одного и того же символа – неоптимально (если мы заменили  $x$  на  $y$ , потом  $y$  на  $z$ , выгоднее было сразу заменить  $x$  на  $z$ ).
- Две замены разных символов можно менять местами.
- Два удаления или два добавления можно менять местами.
- Добавление символа с его последующим удалением – неоптимально (можно отменить оба действия).
- Удаление и добавление разных символов можно менять местами.
- Добавление символа с его последующей заменой – неоптимально (излишняя замена).
- Добавление символа и замена другого символа меняются местами.
- Замена символа с его последующим удалением – не оптимально (излишняя замена).
- Удаление символа и замена другого символа меняются местами.

Пусть  $S_1$  заканчивается на символ «а»,  $S_2$  заканчивается на символ «b». Тогда выполняется одно из следующих утверждений:

1. Символ «а», на который кончается  $S_1$ , в какой-то момент был удален. Сделаем это удаление первой операцией. Тогда мы удалили символ «а», после чего превратили первые  $i-1$  символов  $S_1$  в  $S_2$  (на это потребовалось  $D[i-1, j]$  операций), значит, всего потребовалось  $D[i-1, j] + 1$  операций.
2. Символ «b», на который кончается  $S_2$ , в какой-то момент был добавлен. Сделаем это добавление последней операцией. Мы превратили  $S_1$  в первые  $j-1$  символов  $S_2$ , после чего добавили «b». Аналогично предыдущему случаю, потребовалось  $D[i, j-1] + 1$  операций.

3. Оба предыдущих утверждения неверны. Если мы добавляли символы справа от «а», то чтобы сделать последним символом «b», мы должны были или в какой-то момент добавить его (но тогда утверждение 2 было бы верно), либо заменить на него один из этих добавленных символов (что тоже невозможно, потому что добавление символа с его последующей заменой не оптимально). Значит, символов справа от финального «а» мы не добавляли. самого финального «а» мы не удаляли, поскольку утверждение 1 неверно. Значит, единственный способ изменения последнего символа – его замена. Заменять его 2 или больше раз неоптимально. Следовательно, возможны два варианта:

3.1. Если «а» совпадает с «b», мы последний символ не меняли. Поскольку мы его также не стирали и не приписывали ничего справа от него, он не влиял на наши действия, и, значит, мы выполнили  $D[i - 1, j - 1]$  операций,  $m(S_1[i], S_2[j]) = 0$ .

3.2. Если «а» не совпадает с «b», мы последний символ меняли один раз. Сделаем эту замену первой. В дальнейшем, аналогично предыдущему случаю, мы должны выполнить  $D[i - 1, j - 1]$  операций, значит, всего потребуется  $D[i - 1, j - 1] + 1$  операций,  $m(S_1[i], S_2[j]) = 1$ .

Поскольку расстояние Левенштейна определяет минимальное количество действий, то берется минимум из утверждений 1-3.

Если вычисляется расстояние Дамерау-Левенштейна (транспозиция считается единичным расстоянием), то на основе вычисленных элементов матрицы  $D[i, j]$  вычисляются элементы матрицы  $DT[i, j]$  (формула 2).

$$DT[i, j] = \begin{cases} \min( & ; \text{Если выполняются} \\ D[i, j] \text{ (расстояние Левенштейна)} & \text{условия } i > 1, j > 1, \\ D[i-2, j-2] + m(S_1[i], S_2[j]) & S_1[i] = S_2[j-1], \\ \text{(транспозиция)} & S_1[i-1] = S_2[j] \\ ) & \\ D[i, j] & ; \text{Если условия} \\ & \text{не выполняются} \end{cases} \quad (2)$$

Значение в правом нижнем элементе  $DT[M, N]$  и будет являться расстоянием Дамерау-Левенштейна.

### 10.3 Пример вычисления расстояния Дамерау-Левенштейна

Допустим, что при вводе в справочник информационной системы оператор допустил ряд ошибок при вводе фамилии «ИВАНОВ»:

1. Добавление (I) буквы Н в середине слова – «ИВАННОВ».
2. Удаление (D) буквы И в начале слова – «ВАННОВ».
3. Замена (R) буквы В на Б – «БАННОВ».
4. Перестановка (T) букв В и О – «БАННВО».

Так как каждое действие увеличивает расстояние на 1, то итоговое расстояние Дамерау-Левенштейна равно 4.

Пример вычисления расстояния с помощью матрицы  $DT[i, j]$  приведен на рис. 27.

		<b>И</b>	<b>В</b>	<b>А</b>	<b>Н</b>	<b>О</b>	<b>В</b>
	<b>0</b>	1	2	3	4	5	6
<b>Б</b>	1	1	2	3	4	5	6
<b>А</b>	2	2	2	2	3	4	5
<b>Н</b>	3	3	3	3	2	3	4
<b>Н</b>	4	4	4	4	3	3	4
<b>В</b>	5	5	4	5	4	4	3
<b>О</b>	6	6	5	5	5	4	<b>4</b>

Рис. 27. Пример вычисления расстояния Дамерау-Левенштейна.



В правой нижней ячейке матрицы находится вычисленное расстояние – число 4.

Таким образом, использование расстояния Дамерау-Левенштейна позволяет находить слово даже при большом количестве опечаток.

### **10.4 Алгоритм Вагнера-Фишера вычисления расстояния Дамерау-Левенштейна**

Существует несколько алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

Алгоритм Вагнера-Фишера – это наиболее простой алгоритм, который вычисляет расстояние Дамерау-Левенштейна на основе определения (формула 2).

Недостатком алгоритма является невысокая скорость работы и большие затраты памяти.

Временная сложность алгоритма  $O(M \times N)$ , где  $M$  и  $N$  – длины строк. То есть время выполнения алгоритма прямо пропорционально количеству ячеек в матрице DT.

Рассмотрим реализацию алгоритма на языке C# в соответствии с **примером 16:**

```

/// <summary>
/// Вычисление расстояния Дамерау-Левенштейна
/// </summary>
public static int Distance(string str1Param, string str2Param)
{
    if ((str1Param == null) || (str2Param == null)) return -1;

    int str1Len = str1Param.Length;
    int str2Len = str2Param.Length;

    //Если хотя бы одна строка пустая,
    //возвращается длина другой строки
    if ((str1Len == 0) && (str2Len == 0)) return 0;
    if (str1Len == 0) return str2Len;
    if (str2Len == 0) return str1Len;

    //Приведение строк к верхнему регистру

```

```

string str1 = str1Param.ToUpper();
string str2 = str2Param.ToUpper();

//Объявление матрицы
int[,] matrix = new int[str1Len + 1, str2Len + 1];

//Инициализация нулевой строки и нулевого столбца матрицы
for (int i = 0; i <= str1Len; i++) matrix[i, 0] = i;
for (int j = 0; j <= str2Len; j++) matrix[0, j] = j;

//Вычисление расстояния Дамерау-Левенштейна
for (int i = 1; i <= str1Len; i++)
{
    for (int j = 1; j <= str2Len; j++)
    {
        //Эквивалентность символов, переменная symbEqual
        //соответствует m(s1[i],s2[j])
        int symbEqual = (
            str1.Substring(i - 1, 1) ==
            str2.Substring(j - 1, 1)) ? 0 : 1);

        int ins = matrix[i, j - 1] + 1; //Добавление
        int del = matrix[i - 1, j] + 1; //Удаление
        int subst = matrix[i - 1, j - 1] + symbEqual; //Замена

        //Элемент матрицы вычисляется
        //как минимальный из трех случаев
        matrix[i, j] = Math.Min(Math.Min(ins, del), subst);

        //Дополнение Дамерау по перестановке соседних символов
        if ((i > 1) && (j > 1) &&
            (str1.Substring(i - 1, 1) == str2.Substring(j - 2, 1)) &&
            (str1.Substring(i - 2, 1) == str2.Substring(j - 1, 1)))
        {
            matrix[i, j] = Math.Min(matrix[i, j],
                matrix[i - 2, j - 2] + symbEqual);
        }
    }
}
//Возвращается нижний правый элемент матрицы
return matrix[str1Len, str2Len];
}

```

Функция принимает на вход две строки, вычисляет значения матрицы DT и возвращает значение нижнего правого элемента матрицы.

Вспомогательная функция выводит в консоль результаты вычисления расстояния Дамерау-Левенштейна:

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

```

/// Вывод расстояния Дамерау-Левенштейна в консоль
/// </summary>
public static void WriteDistance(string str1Param, string str2Param)
{
    int d = Distance(str1Param, str2Param);
    Console.WriteLine("'" + str1Param + "', '" +
        str2Param + "' -> " + d.ToString());
}

```

Пример вычисления расстояния:

```

Console.WriteLine("Добавление одного символа в начало середину и
конец строки");
EditDistance.WriteDistance("пример", "1пример");
EditDistance.WriteDistance("пример", "при1мер");
EditDistance.WriteDistance("пример", "пример1");

Console.WriteLine("Добавление двух символов в начало середину и
конец строки");
EditDistance.WriteDistance("пример", "12пример");
EditDistance.WriteDistance("пример", "при12мер");
EditDistance.WriteDistance("пример", "пример12");

Console.WriteLine("Добавление трех символов");
EditDistance.WriteDistance("пример", "1при2мер3");

Console.WriteLine("Транспозиция");
EditDistance.WriteDistance("пРИМер", "прМИер");

Console.WriteLine("Рассмотренный ранее пример");
EditDistance.WriteDistance("ИВАНОВ", "БАННВО");

```

Результаты вывода в консоль:

```

Добавление одного символа в начало середину и конец строки
'пример', '1пример' -> 1
'пример', 'при1мер' -> 1
'пример', 'пример1' -> 1
Добавление двух символов в начало середину и конец строки
'пример', '12пример' -> 2
'пример', 'при12мер' -> 2
'пример', 'пример12' -> 2
Добавление трех символов
'пример', '1при2мер3' -> 3
Транспозиция
'пРИМер', 'прМИер' -> 1
Рассмотренный ранее пример
'ИВАНОВ', 'БАННВО' -> 4

```

## 11 Основы разработки пользовательского интерфейса с использованием технологии Windows Forms

В предыдущих примерах данного пособия создавались консольные приложения. В данном разделе рассмотрим основы создания оконных приложений на основе **примера 17**.

### 11.1 Создание проекта

Для создания оконного приложения необходимо создать в Visual Studio новый проект типа «приложение Windows Forms», что показано на рис. 28.

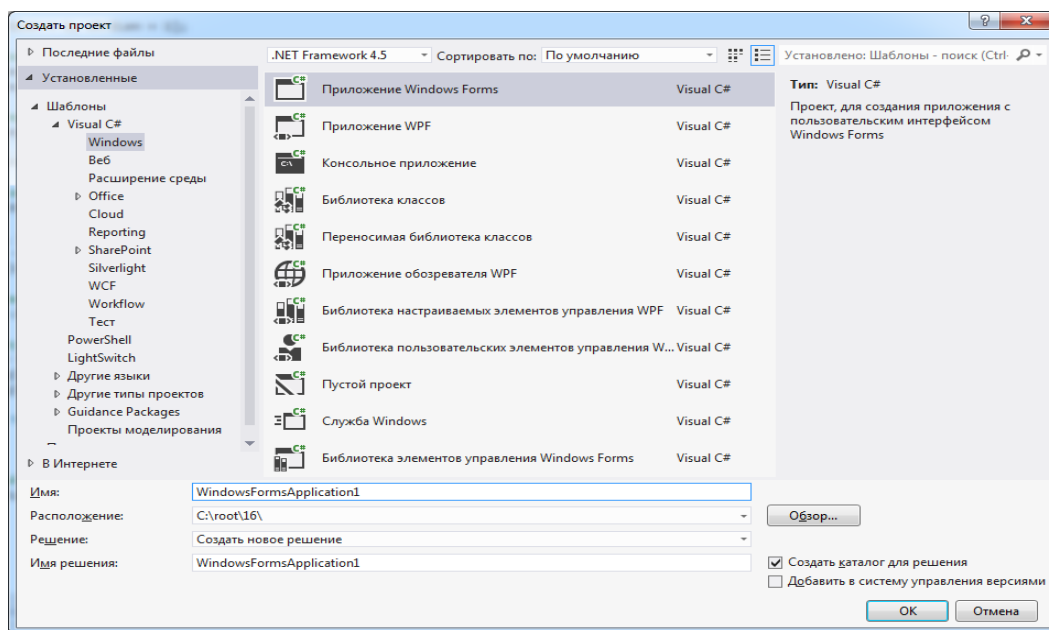


Рис. 28. Создание приложения Windows Forms.

После создания проекта в Visual Studio открывается не окно кода, а редактор макета формы. При этом автоматически создается форма «Form1».

При нажатии на вкладку «Панель элементов» (вкладка расположена в левой части окна) открывается панель, содержащая список элементов,

которые можно перетащить на форму с помощью мыши, что показано на рис. 29.

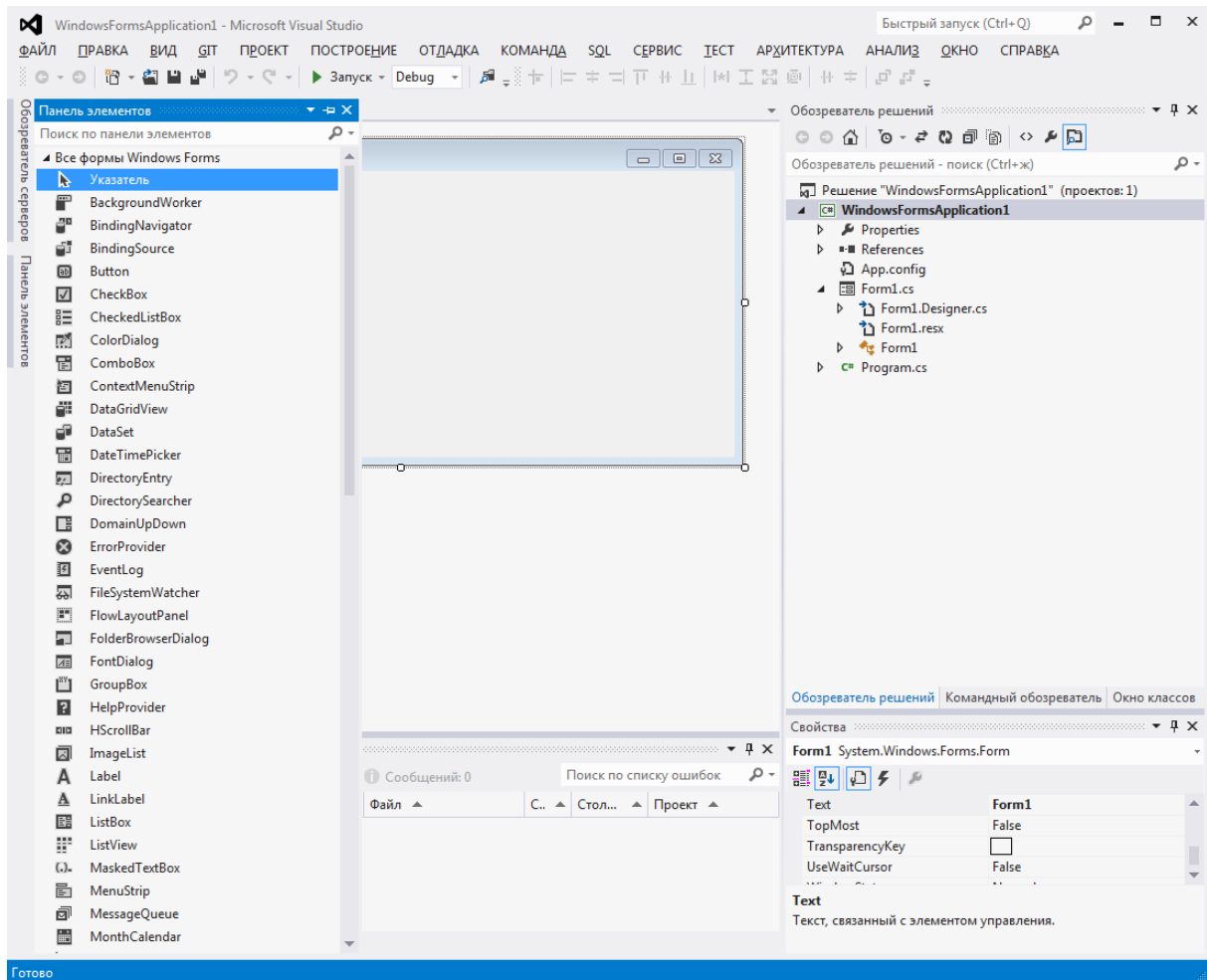


Рис. 29. Пример панели элементов.

Все элементы делятся на визуальные и невидимые.

Визуальные элементы отображаются на форме в виде элементов управления. Это такие элементы как кнопка (Button), поле ввода (TextBox), текстовая надпись (Label).

Невидимые элементы также можно перетащить на форму, однако они отображаются в специальной области редактора макета формы. Примером невидимого элемента управления является таймер (Timer).

Если какая-либо из панелей не видна в текущей настройке Visual Studio, то ее можно включить в пункте меню «ВИД» (в англоязычной версии «View»).

Отметим, что количество элементов составляет несколько десятков и включает самые разнообразные элементы, которые детально рассмотрены в [1]. В том числе элементы из группы «Данные» позволяют достаточно быстро разрабатывать макеты приложений на основе реляционной СУБД.

### **11.2 Пример работы с кнопкой и текстовым полем**

Перетащим в текущее окно элементы Button, TextBox, Label и Timer. Элементы можно свободно перемещать мышью на форме. Отметим, что элемент Timer отображается не на форме, а в отдельной нижней части конструктора. Выделим элемент Label. В правой нижней части окна Visual Studio отображается панель свойств. Результат показан на рис. 30.

В панели свойств отображаются свойства текущего выделенного элемента Label. Изменим свойство «Text» на «Текст по нажатию кнопки».

Для элемента Button изменим свойство «Text» на «Тест».

Отметим, что в данном примере не изменяются стандартные имена элементов управления, они называются «button1», «textBox1» и т.д. Имя элемента управления можно изменить с помощью свойства «Name».

Имена элементов управления рекомендуется менять сразу после перетаскивания на форму, так как имена элементов управления используются в текстах программ.

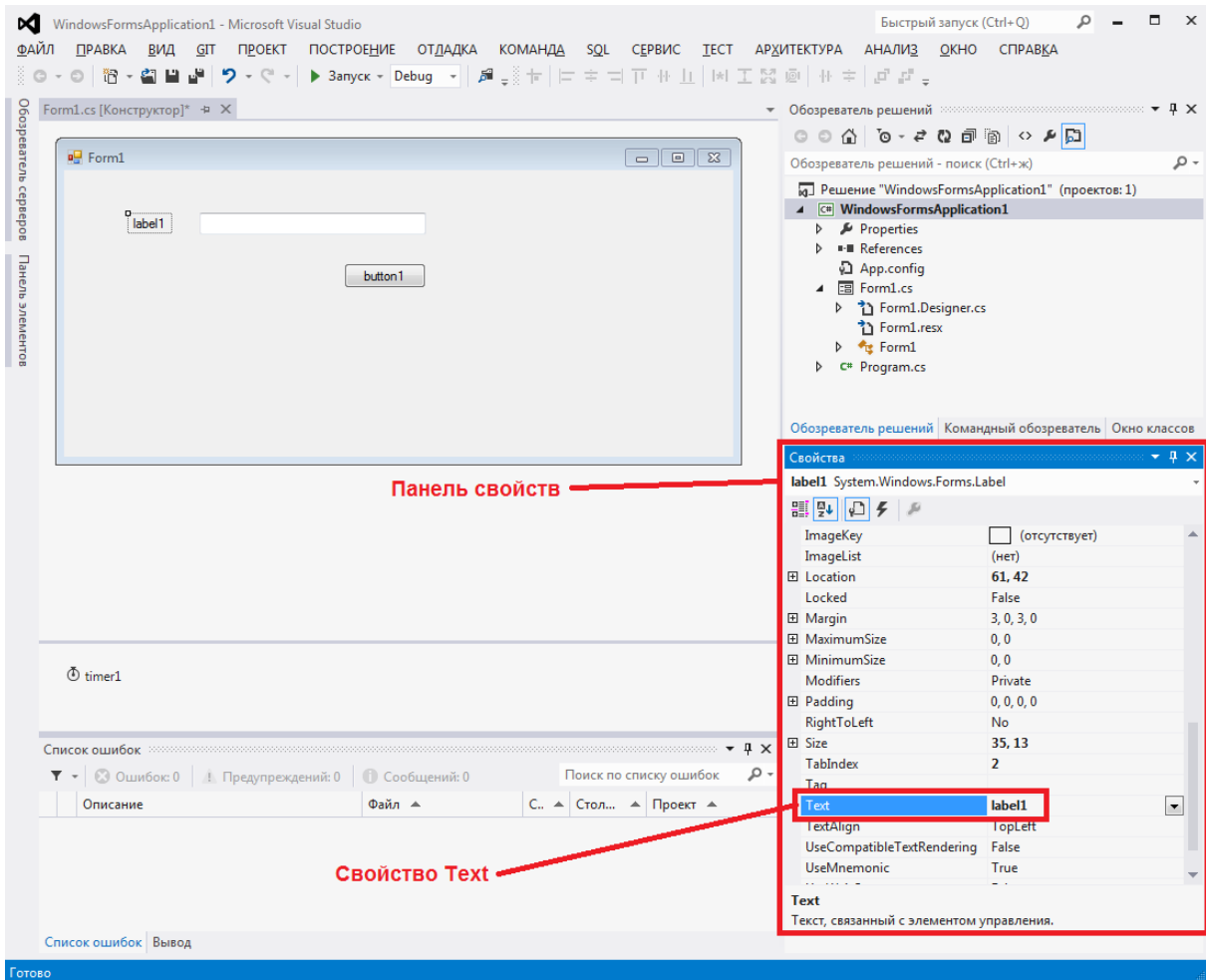






Рис. 30. Пример панели свойств.

В панели свойств используются кнопки, описание которых приведено в таблице 7.

Таблица 7. Кнопки панели свойств.

Кнопка	Описание
	Переключение к списку свойств элемента
	Переключение к списку событий элемента
	Сортировка текущего списка свойств (событий) по категориям
	Сортировка текущего списка свойств (событий) по алфавиту по названиям свойств (событий)

Осуществим двойное нажатие левой клавишей мыши на кнопке Button. При этом будет автоматически сформирован обработчик события нажатия на кнопку (`button1_Click`) и откроется окно редактирования обработчика события, что показано на рис. 31.

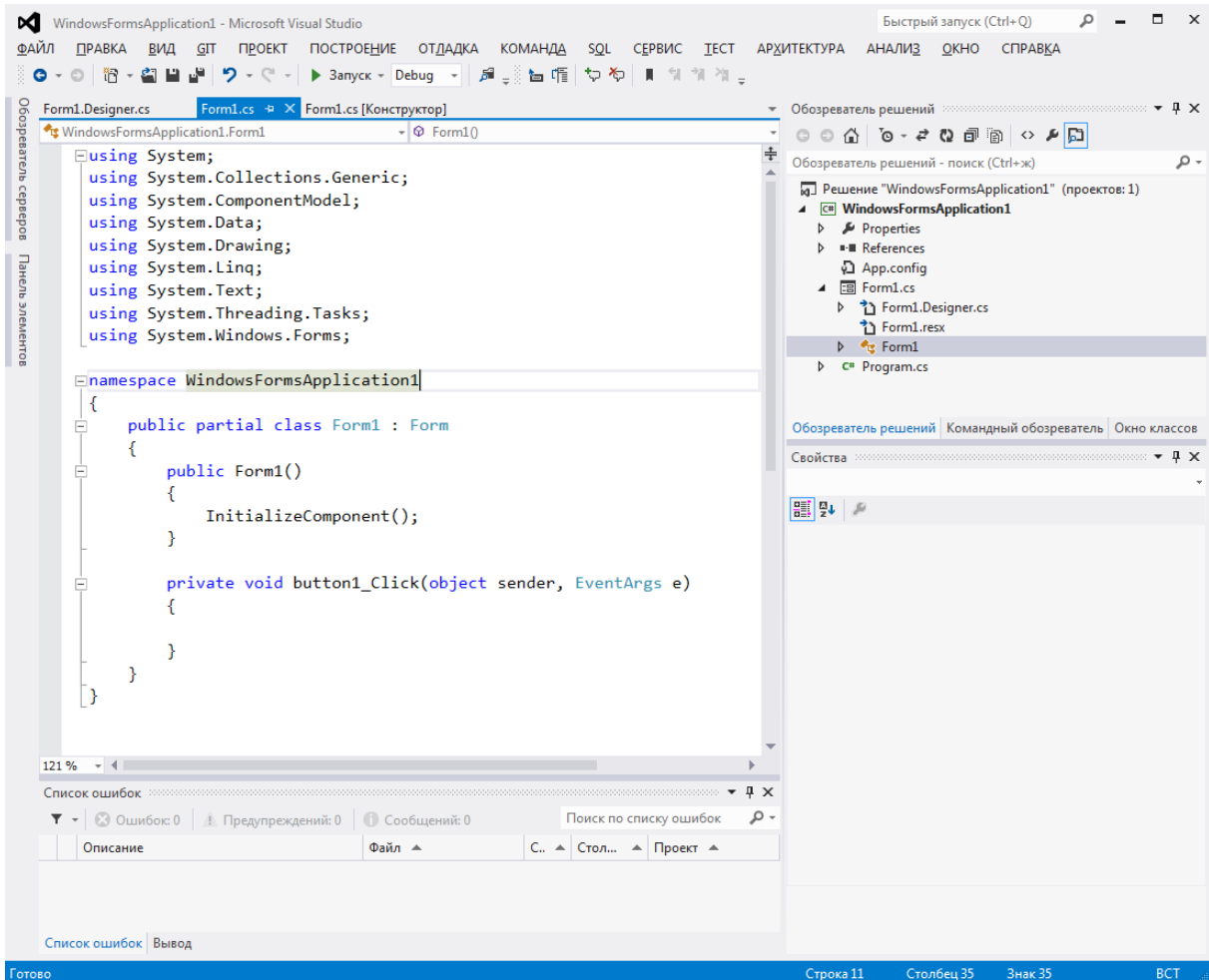


Рис. 31. Обработчик события нажатия на кнопку.

С точки зрения языка C# форма «Form1» представляет собой класс, унаследованный от класса `Form` (расположен в пространстве имен `System.Windows.Forms`).

Обработчики событий добавляются в данный класс как методы.

Отметим, что данный класс объявлен как частичный (`partial`), а в конструкторе вызывается метод «`InitializeComponent`», который в данном файле не объявлен. Значит, в другом файле должна быть еще одна часть данного класса. Если на методе «`InitializeComponent`» нажать правую



кнопку мыши и выбрать пункт меню «Перейти к определению» то будет открыт файл «Form1.Designer.cs», который содержит вторую, автоматически сгенерированную часть класса «Form1». Текст файла «Form1.Designer.cs»:

```
namespace WindowsFormsApplication1
{
    partial class Form1
    {
        /// <summary>
        /// Требуется переменная конструктора.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Освободить все используемые ресурсы.
        /// </summary>
        /// <param name="disposing">истинно, если управляемый ресурс
        должен быть удален; иначе ложно.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Код, автоматически созданный конструктором форм Windows

        /// <summary>
        /// Обязательный метод для поддержки конструктора - не изменяйте
        /// содержимое данного метода при помощи редактора кода.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.button1 = new System.Windows.Forms.Button();
            this.timer1 = new System.Windows.Forms.Timer(this.components);
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.label1 = new System.Windows.Forms.Label();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(289, 74);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "Тест";
            this.button1.UseVisualStyleBackColor = true;
        }
    }
}
```

```

        this.button1.Click += new
System.EventHandler(this.button1_Click);
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(157, 39);
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(207, 20);
        this.textBox1.TabIndex = 1;
        //
        // label1
        //
        this.label1.AutoSize = true;
        this.label1.Location = new System.Drawing.Point(12, 42);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(139, 13);
        this.label1.TabIndex = 2;
        this.label1.Text = "Текст по нажатию кнопки";
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(612, 262);
        this.Controls.Add(this.label1);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
        this.PerformLayout();

    }

    #endregion

    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Timer timer1;
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.Label label1;
}
}

```

В данном файле в виде private-полей объявлены элементы управления, которые были добавлены на форму: button1, timer1, textBox1, label1.

В методе «InitializeComponent» производится задание свойств объектов, которые были установлены в панели свойств: расположение на форме (Location), текстовая подпись (Text) и др.

Обработчик события button1\_Click прикрепляется к настоящему событию, которое расположено в классе Button (точнее в классе Control, от

которого наследуется Button). Таким образом, рассмотренный ранее в пособии механизм событий используется в технологии Windows Forms.

Отметим, что если для элемента управления были созданы свойства и события, а потом он был переименован, то файл «Form1.Designer.cs» не регенерируется. При компиляции это может приводить к ошибкам в файле «Form1.Designer.cs», что требует исправления файла «Form1.Designer.cs» вручную.

Добавим в обработчик события «button1\_Click» следующий код:

```
private void button1_Click(object sender, EventArgs e)
{
    //Запись текста в текстовое поле
    textBox1.Text = "Кнопка нажата";
    //Окно сообщения
    MessageBox.Show("Кнопка нажата");
}
```

При нажатии на кнопку производится запись текста в текстовое поле, используется свойство Text текстового поля textBox1. Отметим, что возможно изменение практически любого свойства элемента, в том числе цвета, шрифта, положения на форме и т.д.

С использованием метода Show статического класса MessageBox осуществляется вывод окна сообщения. Окна сообщений в частности удобны для вывода сообщений об ошибках.

Нажмем на кнопку «Тест». Результат выполнения показан на рис 32.

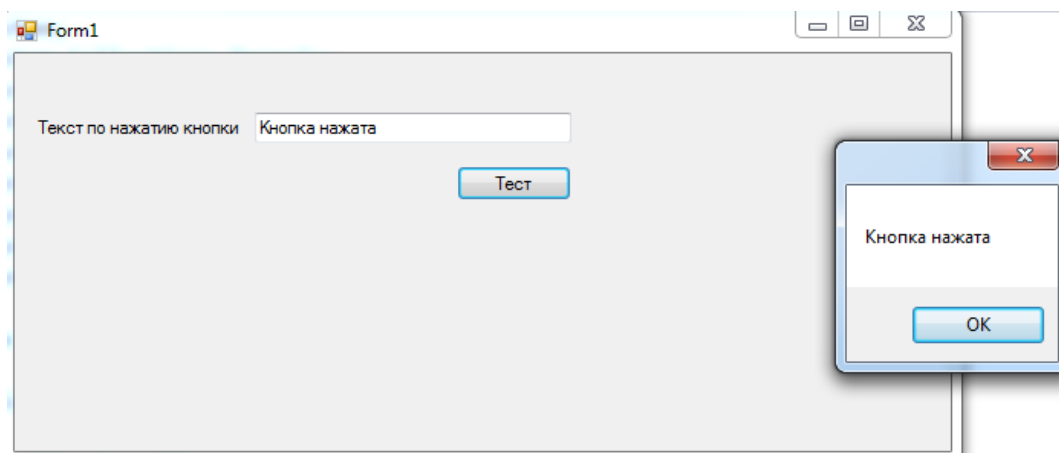


Рис. 32. Обработчик события нажатия на кнопку.

### 11.3 Пример работы с таймером

Добавим на форму следующие элементы (в скобках заданы значения свойств, которые требуется установить):

- Label (Text = «Таймер:»);
- TextBox (Name = «textTimer»);
- Button (Name = «buttonStartTimer», Text = «Старт»);
- Button (Name = «buttonStopTimer», Text = «Стоп»);
- Button (Name = «buttonClearTimer», Text = «Сброс»);
- Timer (Name = «timer1», Interval = «1000»).

Отметим, что элемент Timer с именем timer1 был уже ранее добавлен на форму, для него необходимо только установить свойство Interval.

Общий вид формы в конструкторе форм показан на рис. 33.

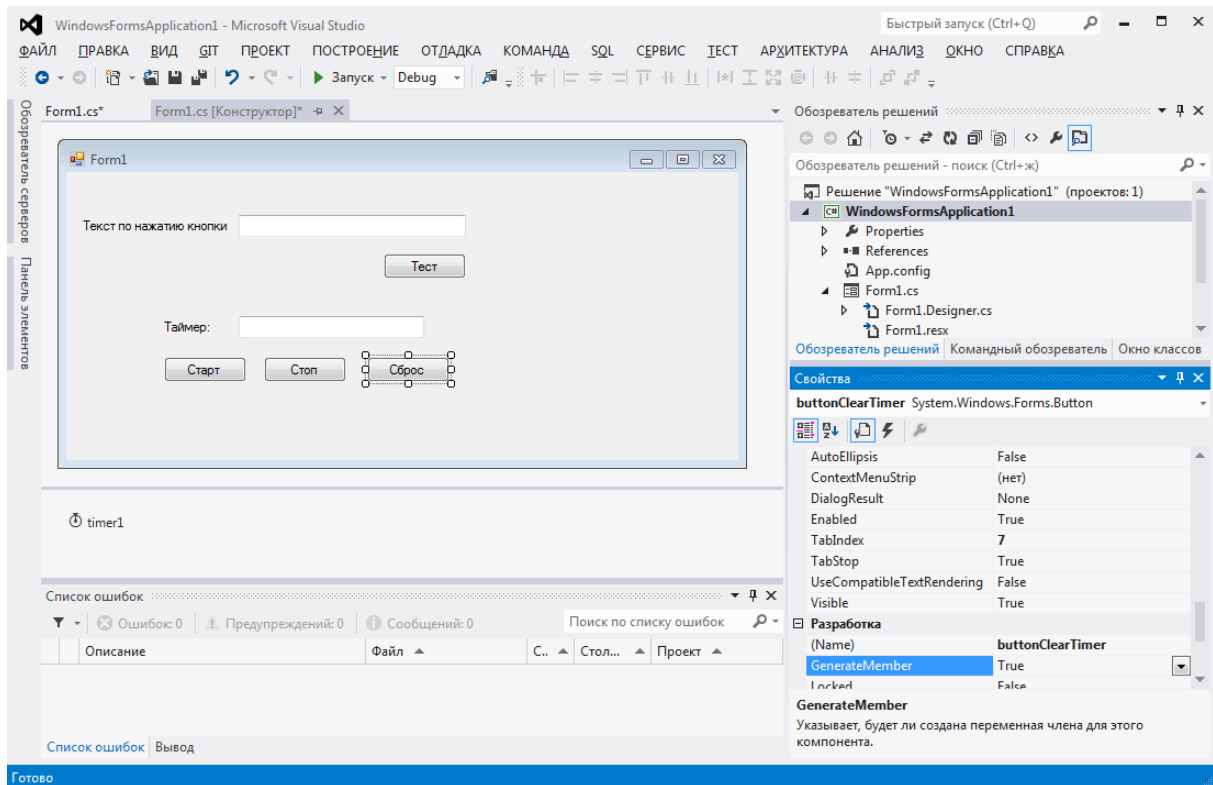


Рис. 33. Форма с добавлением таймера.

Свойство Interval=1000 для таймера означает, что таймер будет срабатывать один раз в секунду (1000 миллисекунд).

Осуществим двойное нажатие левой клавишей мыши на таймере и трех добавленных кнопках, при этом будут сгенерированы соответствующие обработчики событий, которые пока не содержат код:

```
private void timer1_Tick(object sender, EventArgs e)
{
}

private void buttonStartTimer_Click(object sender, EventArgs e)
{
}

private void buttonStopTimer_Click(object sender, EventArgs e)
{
}

private void buttonClearTimer_Click(object sender, EventArgs e)
{
}
```

Отметим, что с точки зрения C# файл с описанием обработчиков событий является обычным классом. Поэтому в него можно добавлять поля данных, свойства, методы.

Однако при разработке приложений считается необходимым реализовывать сложные алгоритмы в виде отдельных классов и из форм осуществлять только их вызов. Это облегчает тестирование алгоритмов и позволяет использовать их в различных приложениях, например в приложении Windows Forms и ASP.NET MVC.

Добавим в файл, содержащий обработчики событий, поле для хранения текущего состояния таймера и метод обновления текущего состояния таймера:

```
/// <summary>
/// Текущее состояние таймера
/// </summary>
(TimeSpan currentTimer = new TimeSpan());

/// <summary>
/// Обновление текущего состояния таймера
/// </summary>
```

```
private void RefreshTimer()
{
    //Обновление поля таймера в форме
    textTimer.Text = currentTimer.ToString();
}
```

Для формы добавим обработчик события по загрузке формы, событие в окне свойств показано на рис. 34.

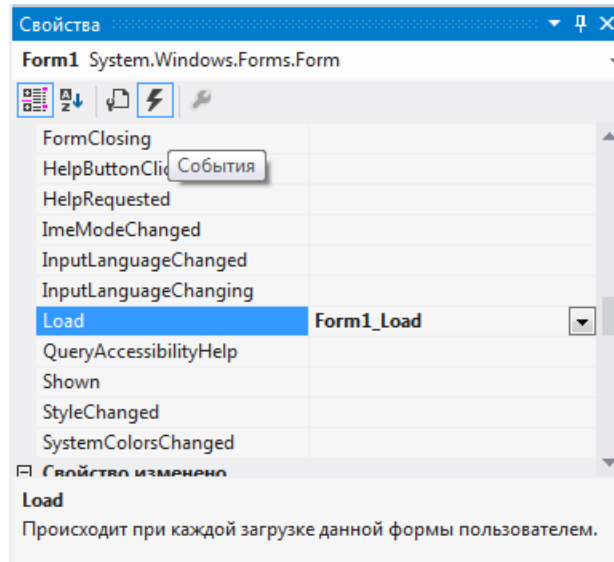


Рис. 34. Обработчик события по загрузке формы.

Для добавления обработчика события необходимо дважды кликнуть на строку «Load».

При загрузке формы выводится начальное (нулевое) значение таймера:

```
private void Form1_Load(object sender, EventArgs e)
{
    //Обновление текущего состояния таймера
    RefreshTimer();
}
```

Отметим, что обработчик события «Load» очень часто используется в формах. В этом обработчике события выполняются действия, которые необходимо выполнить в только что отображенной форме.

Добавим код обработчиков событий таймера и кнопок (каждая строка кода откомментирована):

```
private void timer1_Tick(object sender, EventArgs e)
{
    //Добавление к текущему состоянию таймера
```

```

//интервала в одну секунду
currentTimer = currentTimer.Add(new TimeSpan(0, 0, 1));
//Обновление текущего состояния таймера
RefreshTimer();
}

private void buttonStartTimer_Click(object sender, EventArgs e)
{
    //Запуск таймера
    timer1.Start();
}

private void buttonStopTimer_Click(object sender, EventArgs e)
{
    //Остановка таймера
    timer1.Stop();
}

private void buttonClearTimer_Click(object sender, EventArgs e)
{
    //Остановка таймера
    timer1.Stop();
    //Сброс текущего состояния таймера
    currentTimer = new TimeSpan();
    //Обновление текущего состояния таймера
    RefreshTimer();
}

```

Пример работы таймера показан на рис. 35. При загрузке формы в поле таймера отображаются нулевые значения: «00:00:00». Кнопка «Старт» запускает таймер, кнопка «Стоп» останавливает таймер (но не сбрасывает его значение), кнопка «Сброс» останавливает таймер и сбрасывает его значение.

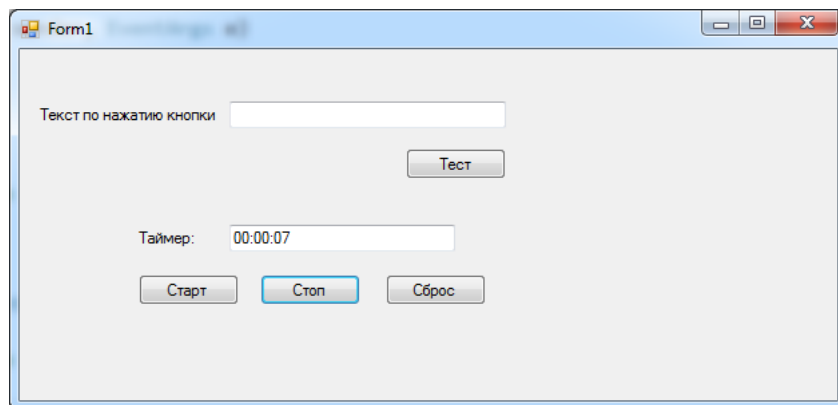


Рис. 35. Пример работы таймера.

## 11.4 Пример открытия дочерних окон

В большинстве случаев приложения Windows Forms содержат более одного окна. В данном разделе мы научимся добавлять и открывать новые окна.

Для добавления нового окна необходимо нажать правую кнопку мыши на проекте и выбрать пункт меню «Добавить/Создать элемент», как показано на рис. 36.

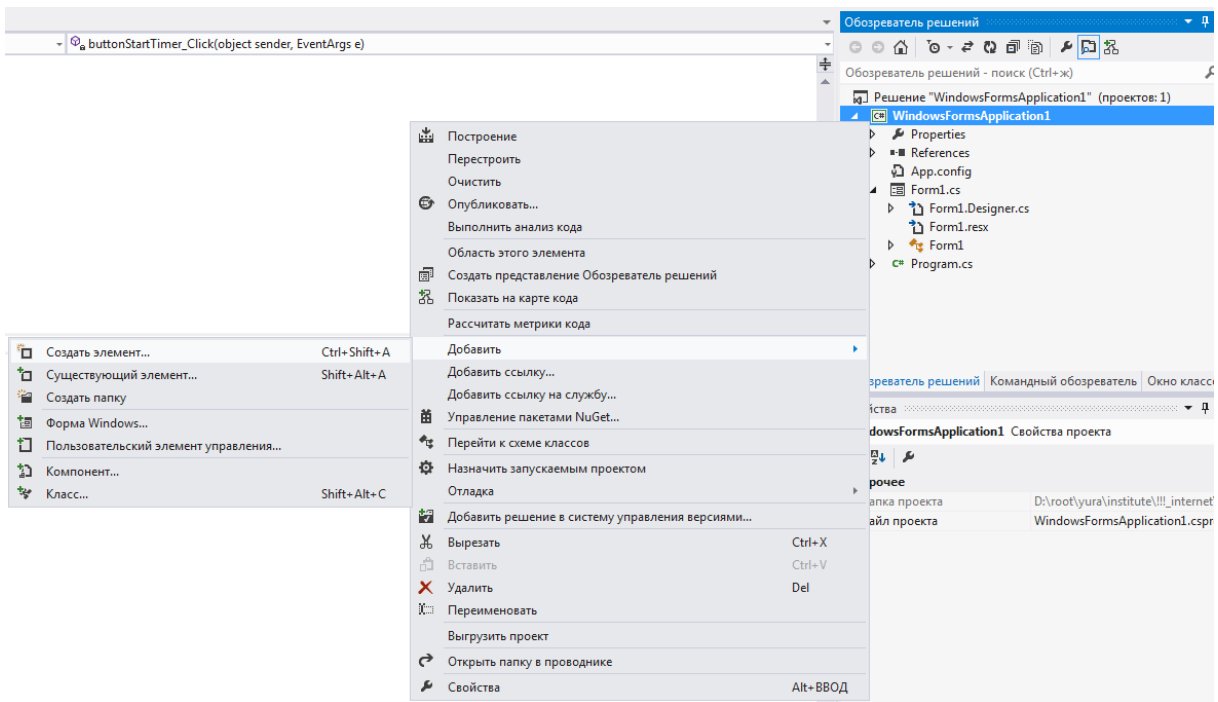


Рис. 36. Добавление формы – шаг 1.

В появившемся меню необходимо выбрать элемент «Форма Windows Forms» и нажать кнопку «Добавить», как показано на рис. 37.

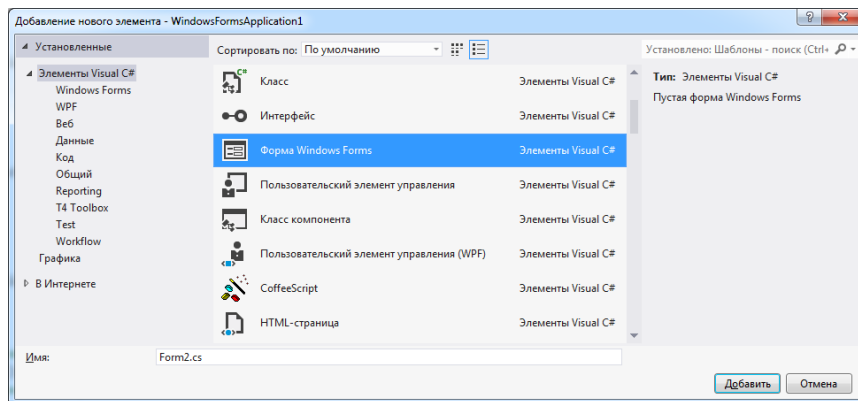


Рис. 37. Добавление формы – шаг 2.



Для добавляемой формы можно указать имя формы (имя класса формы, которое совпадает с именем файла), в данном примере «Form2.cs».

Если приложение содержит несколько форм, то как Windows Forms определяет первую запускаемую форму? Это задается в файле «Program.cs»:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Для приложений Windows Forms метод Main класса Program содержит код для запуска начальной формы. Если в методе «Application.Run» заменить «Form1» на «Form2», то при запуске приложения первой будет открываться «Form2».

Добавим на форму 1 четыре кнопки:

- Button (Name = «buttonOpenNonModal», Text = «Открыть немодальное окно»);
- Button (Name = «buttonOpenModal», Text = «Открыть модальное окно»);
- Button (Name = «buttonClose», Text = «Закреть окно»);
- Button (Name = «buttonExit», Text = «Закреть приложение»).

Внешний вид формы 1 после добавления кнопок показан на рис. 38.

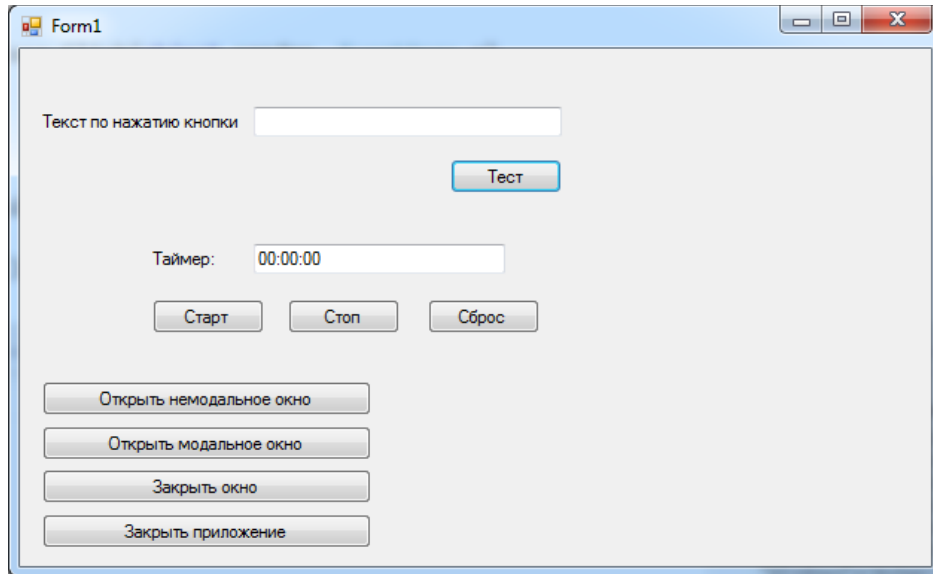


Рис. 38. Добавление кнопок открытия и закрытия окна.

Немодальное окно – это обычное окно приложения, с которым работает пользователь. Если открыто несколько немодальных окон, то между ними возможно переключение фокуса ввода.

Модальное окно – это окно, которое пользователь должен закрыть перед продолжением работы. Если открыто модальное окно, то переключение фокуса ввода в другие окна невозможно. Модальные окна также называют диалоговыми.

Отметим, что рассмотренные ранее окна сообщений, создаваемые с помощью класса `MessageBox`, являются модальными окнами.

Добавим обработчики событий для кнопок.

Кнопка «Открыть немодальное окно»:

```
private void buttonOpenNonModal_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.Show();
}
```

Для открытия дочернего окна необходимо сначала создать новый объект соответствующего класса. При открытии нескольких окон нужно каждый раз создавать новый объект класса, потому что в каждое окно могут быть введены различные данные.

Далее с помощью метода «Show» можно открыть окно в немодальном режиме.

Отметим, что при многократном нажатии на кнопку «Открыть немодальное окно» можно открыть произвольное количество дочерних окон.

Кнопка «Открыть модальное окно»:

```
private void buttonOpenModal_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.ShowDialog();
}
```

С помощью метода «ShowDialog» можно открыть окно в модальном режиме. При этом невозможно вернуться в форму 1 (она не получит фокус) до тех пор, пока не будет закрыта форма 2.

Отметим, что многократное нажатие на кнопку «Открыть модальное окно» невозможно, так как при первом нажатии на кнопку открывается модальное окно, которое необходимо закрыть перед продолжением работы.

Кнопка «Закрыть окно»:

```
private void buttonClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Данная кнопка вызывает метод Close для текущего объекта, что приводит к закрытию окна.

Кнопка «Закрыть приложение»:

```
private void buttonExit_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

Данная кнопка вызывает метод Exit для статического класса Application, что приводит к закрытию приложения. Класс Application отвечает за приложение Windows Forms.

Отметим, что кнопки «Закреть окно» и «Закреть приложение» работают одинаково для главной формы и приводят к закрытию приложения.

Добавим данные кнопки на форму 2. Внешний вид формы 2 после добавления кнопок показан на рис. 39.

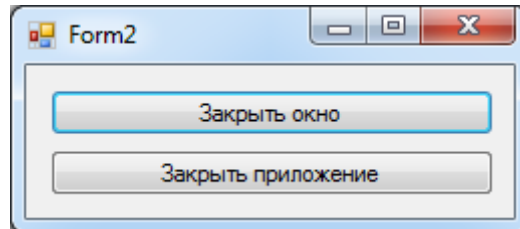


Рис. 39. Добавление кнопок открытия и закрытия окна.

Обработчики событий кнопок совпадают с обработчиками событий в форме 1:

```
private void buttonClose_Click(object sender, EventArgs e)
{
    this.Close();
}
private void buttonExit_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

Однако поведение кнопок в дочерней форме будет отличаться от поведения в родительской форме.

Ключевое слово «this» в данном случае означает текущий объект формы 2. Поэтому при нажатии на кнопку «Закреть окно» будет закрываться форма 2.

Но при нажатии на кнопку «Закреть приложение» будет закрываться все приложение.

Добавим для формы 2 обработчики событий «FormClosing» и «FormClosed», как показано на рис. 40.

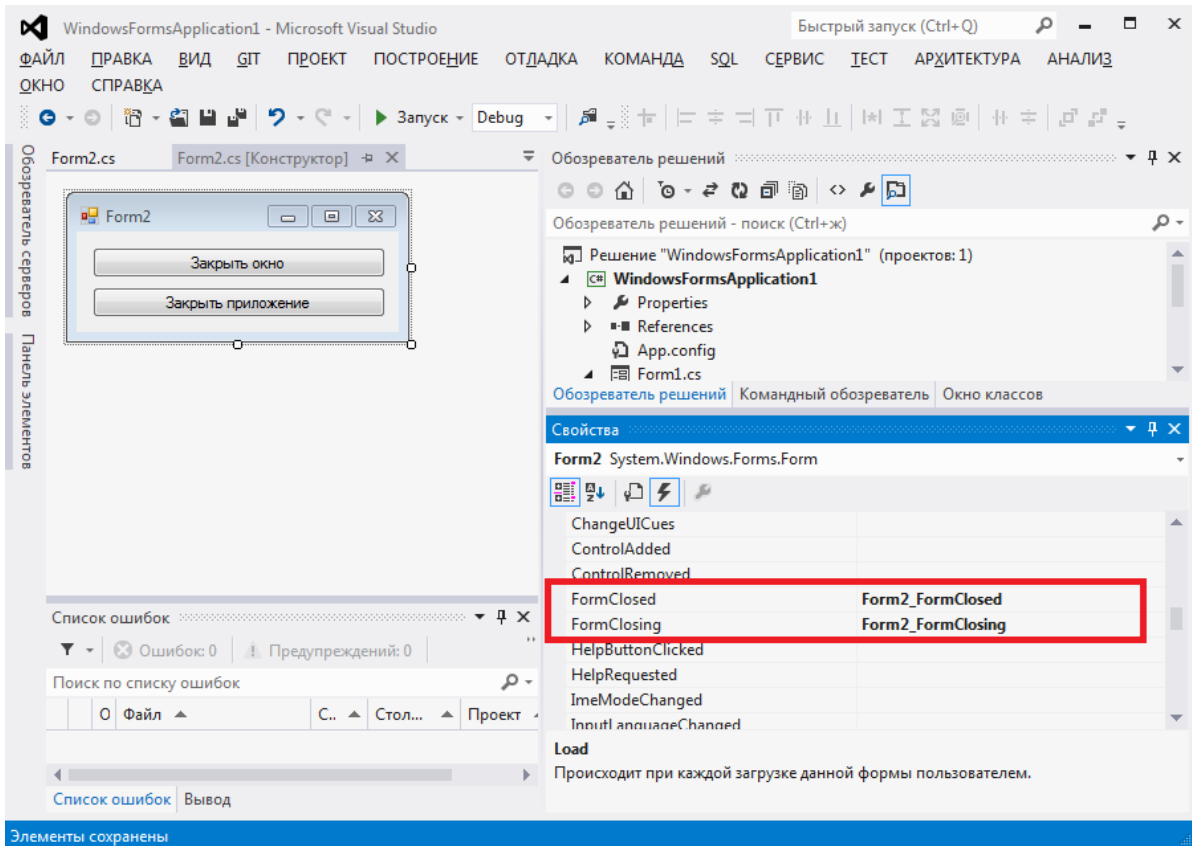


Рис. 40. Обработчики событий FormClosing и FormClosed.

Данные события также довольно часто используются в Windows Forms.

Событие FormClosed возникает, когда окно уже считается закрытым.

Обработчик события:

```
private void Form2_FormClosed(object sender, FormClosedEventArgs e)
{
    MessageBox.Show("Форма 2 закрылась!");
}
```

Событие FormClosing возникает перед закрытием окна, в данном обработчике можно отменить закрытие окна. Обработчик события:

```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
{
    //Вывод диалогового окна
    DialogResult result = MessageBox.Show("Вы действительно хотите закрыть форму 2?", "Уважаемый пользователь", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
    //Отмена закрытия окна
    if (result == DialogResult.No)
    {
```

```

    e.Cancel = true;
}
}

```

В данном примере метод «MessageBox.Show» открывает диалоговое окно в виде вопроса. Пользователь может выбрать кнопки «Да» или «Нет». Если пользователь выбирает кнопку «Нет» то свойство «e.Cancel» (e – параметр метода) устанавливается в true, что приводит к отмене стандартного обработчика события и окно не закрывается.

Результаты работы показаны на рис. 41 и 42.

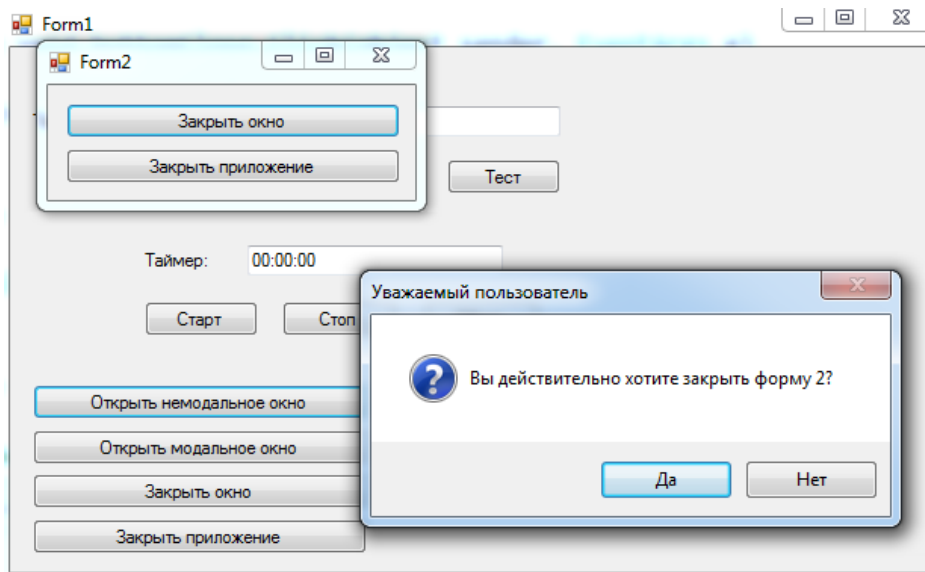


Рис. 41. Результат работы обработчика событий FormClosing.

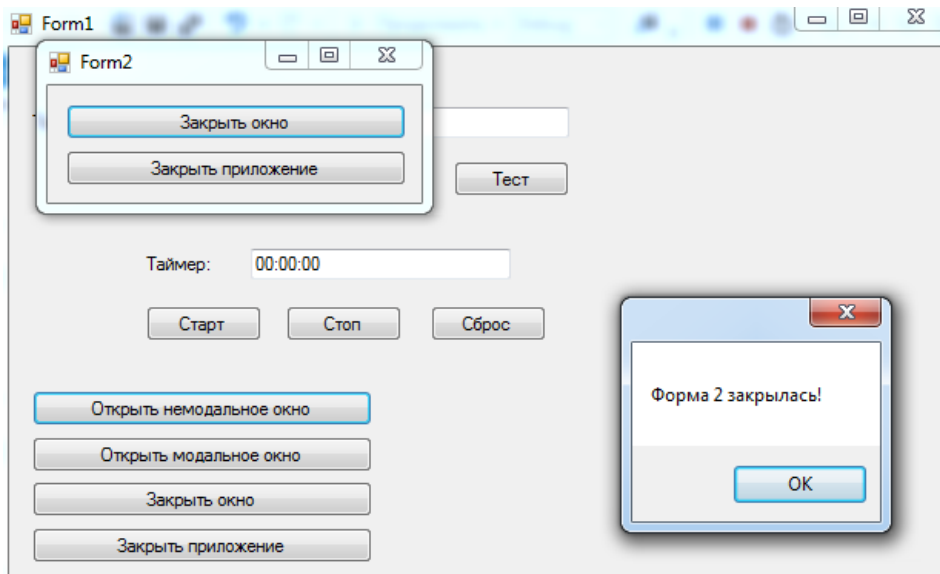


Рис. 42. Результат работы обработчика событий FormClosed.

## 12 Пример многопоточного поиска в текстовом файле с использованием технологии Windows Forms

В завершающем разделе пособия мы реализуем комплексный пример, который обобщает примеры, рассмотренные в предыдущих разделах.

Пример реализован с использованием технологии Windows Forms. Он выполняет следующие действия:

- Запрашивает имя текстового файла для поиска с использованием стандартного диалога открытия файла.
- Осуществляет ввод слова для поиска.
- Реализует четкий (без учета опечаток) поиск слова в файле. Найденное слово (которое может встретиться в файле несколько раз) выводится в результирующий список.
- Реализует нечеткий (на основе расстояния Дамерау-Левенштейна) многопоточный поиск в файле. При этом вводится максимальное расстояние для нечеткого поиска, на которое искомое слово может отличаться от слова в файле. Также вводится количество потоков, на которое разделяется массив слов исходного файла. Найденное слово (которое может встретиться в файле несколько раз) выводится в результирующий список с указанием номера потока и вычисленного расстояния Дамерау-Левенштейна.
- По результатам поиска формируется отчет в виде текстового файла в формате HTML. Выбор имени файла при сохранении производится с использованием стандартного диалога сохранения файла.

Данное приложение реализовано в **примере 18**.

Приложение реализовано с использованием одной формы, которая показана на рис. 43.

Рис. 43. Форма многопоточного поиска в текстовом файле.

### 12.1 Чтение информации из текстового файла

При нажатии на кнопку «Чтение из файла» производится чтение информации из текстового файла.

Для хранения списка слов, прочитанных из файла, в классе «Form1» используется поле «list»:

```

/// <summary>
/// Список слов
/// </summary>
List<string> list = new List<string>();

```

Рассмотрим код обработчика кнопки:

```

private void buttonLoadFile_Click(object sender, EventArgs e)
{
    OpenFileDialog fd = new OpenFileDialog();
    fd.Filter = "текстовые файлы|*.txt";
}

```



```

if (fd.ShowDialog() == DialogResult.OK)
{
    Stopwatch t = new Stopwatch();
    t.Start();

    //Чтение файла в виде строки
    string text = File.ReadAllText(fd.FileName);

    //Разделительные символы для чтения из файла
    char[] separators =
        new char[] { ' ', '.', ',', '!', '?', '/', '\t', '\n' };

    string[] textArray = text.Split(separators);

    foreach (string strTemp in textArray)
    {
        //Удаление пробелов в начале и конце строки
        string str = strTemp.Trim();
        //Добавление строки в список, если строка не содержится
в списке
        if (!list.Contains(str)) list.Add(str);
    }

    t.Stop();
    this.textBoxFileReadTime.Text = t.Elapsed.ToString();
    this.textBoxFileReadCount.Text = list.Count.ToString();
}
else
{
    MessageBox.Show(«Необходимо выбрать файл»);
}
}

```

Для выбора текстового файла используется класс `OpenFileDialog`. Данный класс реализует стандартный диалог открытия файла. Свойство «`Filter`» содержит список расширений файлов, которые позволяет выбирать диалоговое окно. Открытие диалогового окна производится с помощью метода «`ShowDialog`».

Если пользователь не выбрал текстовый файл (условие «`fd.ShowDialog() == DialogResult.OK`» не выполняется), то с помощью класса `MessageBox` выводится сообщение «Необходимо выбрать файл» и обработчик события завершает работу.

Если условие истинно (файл успешно выбран), то выполняются основные действия обработчика события.

С использованием класса «Stopwatch» (который был рассмотрен в разделе пособия по параллельной обработке) объявляется и запускается таймер.

С использованием метода «File.ReadAllText» (который был рассмотрен в разделе пособия по обработке текстовых файлов) содержимое файла считывается в виде переменной «text» типа string. Свойство «fd.FileName» класса OpenFileDialog возвращает путь и имя файла.

С использованием метода «Split» класса string производится разделение содержимого переменной «text» на массив строк «textArray». Метод «Split» принимает в качестве параметра массив символов, которые могут разделять слова в файле.

Далее в цикле foreach производится перебор всех слов в массиве «textArray». Для текущего слова производится удаление пробелов в начале и конце. Если текущее слово еще не содержится в списке слов «list», то оно добавляется в этот список. Таким образом, каждое слово в файле включается в список «list» только один раз.

После этого производится остановка таймера. В текстовое поле «textBoxFileReadTime» выводится время построения списка слов файла, а в текстовое поле «textBoxFileReadCount» выводится количество слов в списке «list», то есть количество уникальных слов в файле. Отметим, что на рис. 43. у текстовых полей «textBoxFileReadTime» и «textBoxFileReadCount» серый фон, так как для них установлено свойство «ReadOnly=true», то есть они доступны только для чтения. Если поле доступно только для чтения, то пользователь не может вводить данные в такое поле, но его можно изменять в программе.

## 12.2 Четкий поиск в текстовом файле

В процедуре поиска используется не текстовый файл напрямую, а список «list» слов, прочитанных из файла.

Рассмотрим код обработчика кнопки «Четкий поиск»:

```
private void buttonExact_Click(object sender, EventArgs e)
{
    //Слово для поиска
    string word = this.textBoxFind.Text.Trim();

    //Если слово для поиска не пусто
    if (!string.IsNullOrEmpty(word) && list.Count > 0)
    {
        //Слово для поиска в верхнем регистре
        string wordUpper = word.ToUpper();

        //Временные результаты поиска
        List<string> tempList = new List<string>();

        Stopwatch t = new Stopwatch();
        t.Start();

        foreach (string str in list)
        {
            if (str.ToUpper().Contains(wordUpper))
            {
                tempList.Add(str);
            }
        }

        t.Stop();
        this.textBoxExactTime.Text = t.Elapsed.ToString();

        this.listBoxResult.BeginUpdate();

        //Очистка списка
        this.listBoxResult.Items.Clear();

        //Вывод результатов поиска
        foreach (string str in tempList)
        {
            this.listBoxResult.Items.Add(str);
        }
        this.listBoxResult.EndUpdate();
    }
    else
    {
```

```

        MessageBox.Show("Необходимо выбрать файл и ввести слово для
поиска");
    }
}

```

В начале работы обработчика проверяется содержимое поля «Слово для поиска» (элемент `textBoxFind`).

Если данное поле не пусто «`string.IsNullOrEmpty(word)`» и прочитаны данные из файла «`list.Count > 0`» то выполняются основные действия обработчика события. Если условие не выполняется, то выводится сообщение «Необходимо выбрать файл и ввести слово для поиска» и обработчик события завершает работу.

Далее перебираются все слова в списке «`list`». Если слово в списке «`list`» содержит слово для поиска (что проверяется с использованием метода «`Contains`» класса `string`), то данное слово добавляется во временный список найденных слов «`tempList`». Перед проверкой строки переводятся в верхний регистр с использованием метода «`ToUpper`» класса `string`.

После завершения поиска в текстовое поле «`textBoxExactTime`» выводится время поиска, а в список `listBoxResult` (элемент `ListBox`) список найденных слов.

Для заполнения данными элемента `ListBox` необходимо выполнить следующие действия:

1. Для начала обновления данных списка необходимо вызвать метод «`BeginUpdate`».
2. Для работы со списком необходимо использовать коллекцию «`Items`». Для очистки результатов предыдущего поиска необходимо вызвать метод «`Items.Clear`».
3. Для добавления элементов в список необходимо использовать метод «`Items.Add`».
4. Для завершения обновления данных списка необходимо вызвать метод «`EndUpdate`».

В результате четкого поиска в список будут выведены слова файла, которые включают искомое слово как подстроку.

Пример результатов четкого поиска представлен на рис. 44. В качестве файла используется текстовое содержимое статьи из Википедии о языке программирования C#. Отметим, что для корректного поиска в русскоязычном тексте, текстовый файл должен быть сохранен в кодировке UTF-8, так как с этой кодировкой по умолчанию работают методы класса File.

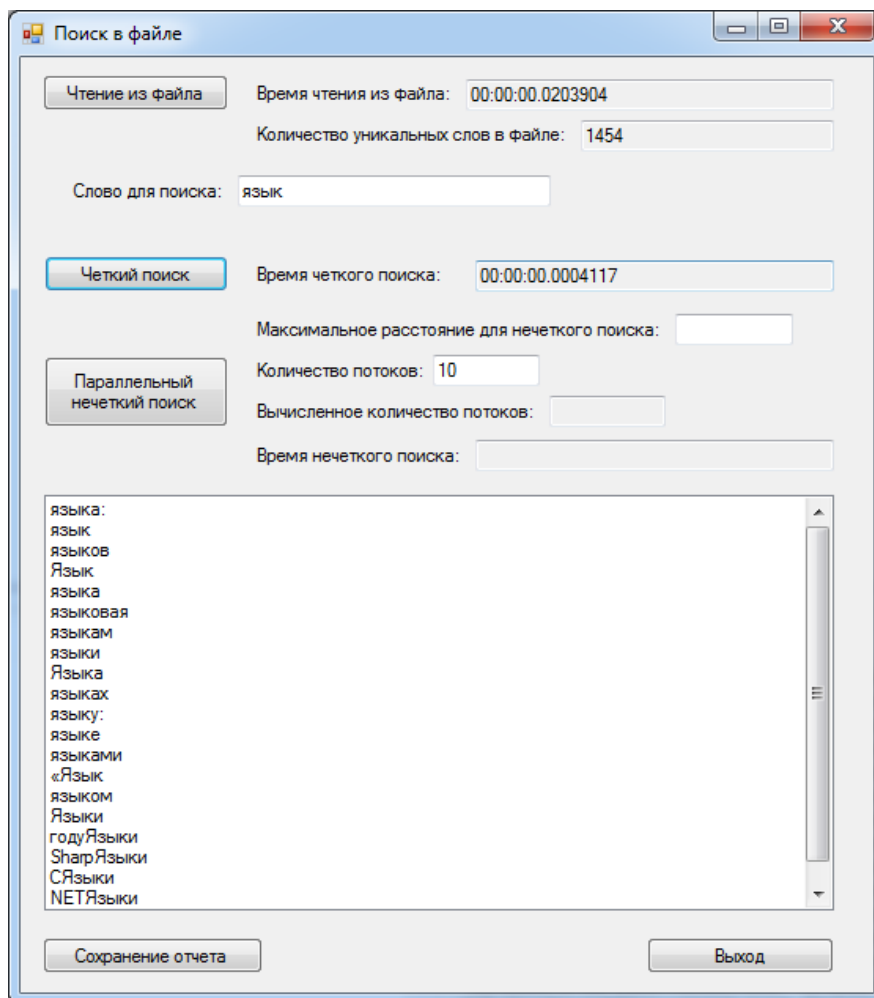


Рис. 44. Результаты четкого поиска.

### **12.3 Нечеткий поиск в текстовом файле**

Нечеткий поиск в текстовом файле практически не отличается от четкого поиска, только вместо проверки вхождения подстроки необходимо

вызывать функцию вычисления расстояния Дамерау-Левенштейна (рассмотренную ранее в пособии).

Однако, дополнительную сложность придает требование параллельного поиска в массиве слов (пример параллельного поиска в массиве также был рассмотрен ранее).

Рассмотрим код обработчика кнопки «Параллельный нечеткий поиск»:

```
private void buttonApprox_Click(object sender, EventArgs e)
{
    //Слово для поиска
    string word = this.textBoxFind.Text.Trim();

    //Если слово для поиска не пусто
    if (!string.IsNullOrEmpty(word) && list.Count > 0)
    {
        int maxDist;
        if (!int.TryParse(this.textBoxMaxDist.Text.Trim(), out
maxDist))
        {
            MessageBox.Show("Необходимо указать максимальное
расстояние");
            return;
        }

        if (maxDist < 1 || maxDist > 5)
        {
            MessageBox.Show("Максимальное расстояние должно быть в
диапазоне от 1 до 5");
            return;
        }

        int ThreadCount;
        if (!int.TryParse(this.textBoxThreadCount.Text.Trim(), out
ThreadCount))
        {
            MessageBox.Show("Необходимо указать количество
потоков");
            return;
        }

        Stopwatch timer = new Stopwatch();
        timer.Start();

        //-----
        // Начало параллельного поиска
```

```

//-----

//Результирующий список
List<ParallelSearchResult> Result = new
List<ParallelSearchResult>();

//Деление списка на фрагменты для параллельного запуска в
потоках
List<MinMax> arrayDivList = SubArrays.DivideSubArrays(0,
list.Count, ThreadCount);
int count = arrayDivList.Count;

//Количество потоков соответствует количеству фрагментов
массива
Task<List<ParallelSearchResult>>[] tasks = new
Task<List<ParallelSearchResult>>[count];

//Запуск потоков
for (int i = 0; i < count; i++)
{
    //Создание временного списка, чтобы потоки не работали
параллельно с одной коллекцией
    List<string> tempTaskList =
list.GetRange(arrayDivList[i].Min, arrayDivList[i].Max -
arrayDivList[i].Min);

    tasks[i] = new Task<List<ParallelSearchResult>>(
        //Метод, который будет выполняться в потоке
        ArrayThreadTask,
        //Параметры потока
        new ParallelSearchThreadParam()
        {
            tempList = tempTaskList,
            maxDist = maxDist,
            ThreadNum = i,
            wordPattern = word
        });

    //Запуск потока
    tasks[i].Start();
}

Task.WaitAll(tasks);

timer.Stop();

//Объединение результатов
for (int i = 0; i < count; i++)
{
    Result.AddRange(tasks[i].Result);
}

```

```

}

//-----
// Завершение параллельного поиска
//-----

timer.Stop();

//Вывод результатов

//Время поиска
this.textBoxApproxTime.Text = timer.Elapsed.ToString();

//Вычисленное количество потоков
this.textBoxThreadCountAll.Text = count.ToString();

//Начало обновления списка результатов
this.listBoxResult.BeginUpdate();

//Очистка списка
this.listBoxResult.Items.Clear();

//Вывод результатов поиска
foreach (var x in Result)
{
    string temp = x.word + "(расстояние=" +
x.dist.ToString() + " поток=" + x.ThreadNum.ToString() + ")";
    this.listBoxResult.Items.Add(temp);
}

//Окончание обновления списка результатов
this.listBoxResult.EndUpdate();
}
else
{
    MessageBox.Show("Необходимо выбрать файл и ввести слово для
поиска");
}
}

```

В начале обработчика, как и в случае четкого поиска, проверяется что список «list» не пуст и что указано слово для поиска. Если условие выполняется, то производится проверка и приведение к типу int полей «Максимальное расстояние для нечеткого поиска» и «Количество потоков». Для приведения к типу int используется метод «int.TryParse».



Для максимального расстояния также проверяется, что оно находится в диапазоне от 1 до 5.

Параллельный поиск практически полностью повторяет пример, рассмотренный в разделе пособия по параллельному поиску. Разница состоит в том, что в данном случае обрабатывается не массив целых чисел, а массив строк.

Для хранения информации о найденных словах используется класс `ParallelSearchResult`:

```

/// <summary>
/// Результаты параллельного поиска
/// </summary>
public class ParallelSearchResult
{
    /// <summary>
    /// Найденное слово
    /// </summary>
    public string word { get; set; }

    /// <summary>
    /// Расстояние
    /// </summary>
    public int dist { get; set; }

    /// <summary>
    /// Номер потока
    /// </summary>
    public int ThreadNum { get; set; }
}

```

Класс содержит найденное слово, расстояние Дамерау-Левенштейна (на которое данное слово отличается от искомого) и номер потока, в котором было найдено данное слово.

Для деления массива на подмассивы используется рассмотренный в разделе параллельного поиска класс `SubArrays`.

Для параллельного поиска используется класс `Task`. Как и в рассмотренном ранее примере параллельного поиска создается массив объектов класса `Task`, потоки запускаются, ожидание завершения потоков осуществляется с помощью метода «`Task.WaitAll`». Внутри потока

выполняется метод `ArrayThreadTask`, который получает в качестве параметра объект класса `ParallelSearchThreadParam`.

После завершения работы всех потоков производится объединение результатов (с использованием свойства `Result` класса `Task`).

Далее осуществляется вывод результатов. Заполняются поля «Вычисленное количество потоков» (так как класс `SubArrays` может возвращать на один поток больше чем указано) и «Время нечеткого поиска».

Аналогично четкому поиску найденные слова выводятся в список «`listBoxResult`», но в данном случае кроме найденного слова выводится номер потока, в котором было найдено слово, и реальное расстояние Дамерау-Левенштейна, на которое данное слово отличается от искомого.

Рассмотрим метод «`ArrayThreadTask`», который выполняется внутри потока.

В качестве параметра метод «`ArrayThreadTask`» получает объект класса «`ParallelSearchThreadParam`»:

```

/// <summary>
/// Параметры которые передаются в поток для параллельного
поиска
/// </summary>
class ParallelSearchThreadParam
{
    /// <summary>
    /// Массив для поиска
    /// </summary>
    public List<string> tempList { get; set; }

    /// <summary>
    /// Слово для поиска
    /// </summary>
    public string wordPattern { get; set; }

    /// <summary>
    /// Максимальное расстояние для нечеткого поиска
    /// </summary>
    public int maxDist { get; set; }

    /// <summary>
    /// Номер потока

```

```

    /// </summary>
    public int ThreadNum { get; set; }
}

```

Класс содержит входной массив слов и слово для поиска, максимальное расстояние для нечеткого поиска и номер потока.

Текст метода «ArrayThreadTask»:

```

/// <summary>
/// Выполняется в параллельном потоке для поиска строк
/// </summary>
public static List<ParallelSearchResult> ArrayThreadTask(object
paramObj)
{
    ParallelSearchThreadParam param =
    (ParallelSearchThreadParam)paramObj;

    //Слово для поиска в верхнем регистре
    string wordUpper = param.wordPattern.Trim().ToUpper();

    //Результаты поиска в одном потоке
    List<ParallelSearchResult> Result = new
    List<ParallelSearchResult>();

    //Перебор всех слов во временном списке данного потока
    foreach (string str in param.tempList)
    {
        //Вычисление расстояния Дамерау-Левенштейна
        int dist = EditDistance.Distance(str.ToUpper(), wordUpper);

        //Если расстояние меньше порогового, то слово добавляется в
результат
        if (dist <= param.maxDist)
        {
            ParallelSearchResult temp = new ParallelSearchResult()
            {
                word = str,
                dist = dist,
                ThreadNum = param.ThreadNum
            };

            Result.Add(temp);
        }
    }
    return Result;
}

```

В методе осуществляется перебор всех слов в массиве для поиска, который был передан данному потоку. Если расстояние Дамерау-

Левенштейна между текущим словом и искомым словом меньше максимально допустимого, то текущее слово считается найденным и помещается в список результата.

Список результата, который является возвращаемым значением метода, содержит объекты рассмотренного ранее класса «ParallelSearchResult».

Пример результатов нечеткого поиска представлен на рис. 45.

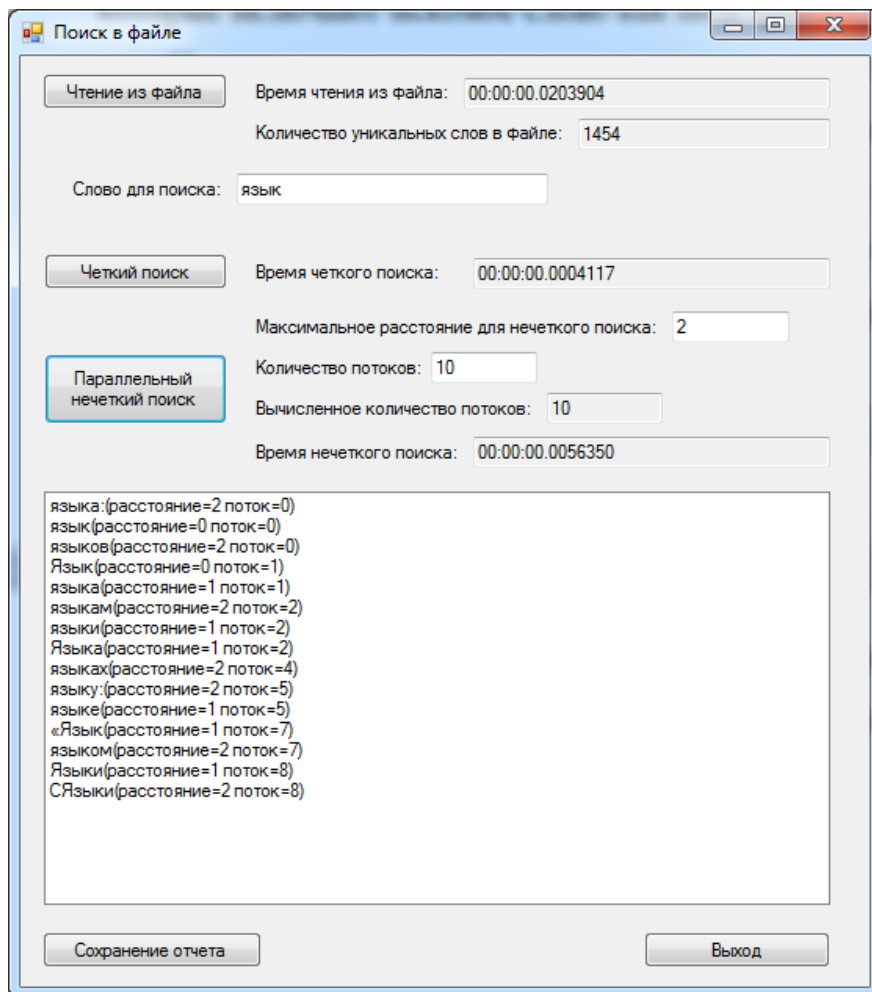


Рис. 45. Результаты нечеткого поиска.

## 12.4 Формирование отчета

При нажатии на кнопку «Сохранение отчета» производится формирование отчета и сохранение в текстовый файл.

Рассмотрим код обработчика кнопки:

```
private void buttonSaveReport_Click(object sender, EventArgs e)
```

```

{
    //Имя файла отчета
    string TempReportFileName = "Report_" +
DateTime.Now.ToString("dd_MM_yyyy_hhmmss");

    //Диалог сохранения файла отчета
    SaveFileDialog fd = new SaveFileDialog();
    fd.FileName = TempReportFileName;
    fd.DefaultExt = ".html";
    fd.Filter = "HTML Reports|*.html";

    if (fd.ShowDialog() == DialogResult.OK)
    {
        string ReportFileName = fd.FileName;

        //Формирование отчета
        StringBuilder b = new StringBuilder();
        b.AppendLine("<html>");

        b.AppendLine("<head>");
        b.AppendLine("<meta http-equiv='Content-Type'
content='text/html; charset=UTF-8'/>");
        b.AppendLine("<title>" + "Отчет: " + ReportFileName +
"</title>");
        b.AppendLine("</head>");

        b.AppendLine("<body>");

        b.AppendLine("<h1>" + "Отчет: " + ReportFileName + "</h1>");
        b.AppendLine("<table border='1'>");

        b.AppendLine("<tr>");
        b.AppendLine("<td>Время чтения из файла</td>");
        b.AppendLine("<td>" + this.textBoxFileReadTime.Text +
"</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr>");
        b.AppendLine("<td>Количество уникальных слов в файле</td>");
        b.AppendLine("<td>" + this.textBoxFileReadCount.Text +
"</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr>");
        b.AppendLine("<td>Слово для поиска</td>");
        b.AppendLine("<td>" + this.textBoxFind.Text + "</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr>");

```

```

        b.AppendLine("<td>Максимальное расстояние для нечеткого
поиска</td>");
        b.AppendLine("<td>" + this.textBoxMaxDist.Text + "</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr>");
        b.AppendLine("<td>Время четкого поиска</td>");
        b.AppendLine("<td>" + this.textBoxExactTime.Text + "</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr>");
        b.AppendLine("<td>Время нечеткого поиска</td>");
        b.AppendLine("<td>" + this.textBoxApproxTime.Text +
"</td>");
        b.AppendLine("</tr>");

        b.AppendLine("<tr valign='top'>");
        b.AppendLine("<td>Результаты поиска</td>");
        b.AppendLine("<td>");
        b.AppendLine("<ul>");

        foreach (var x in this.listBoxResult.Items)
        {
            b.AppendLine("<li>" + x.ToString() + "</li>");
        }

        b.AppendLine("</ul>");
        b.AppendLine("</td>");
        b.AppendLine("</tr>");

        b.AppendLine("</table>");

        b.AppendLine("</body>");
        b.AppendLine("</html>");

        //Сохранение файла
        File.AppendAllText(ReportFileName, b.ToString());

        MessageBox.Show("Отчет сформирован. Файл: " +
ReportFileName);
    }
}

```

В переменной «TempReportFileName» формируется имя файла на основе текущих даты и времени.

Далее создается объект класса SaveFileDialog. При вызове метода «ShowDialog» пользователю предлагается сохранить файл.

Если пользователь не выбрал корректное имя файла для сохранения (условие «fd.ShowDialog() == DialogResult.OK» не выполняется), то обработчик события завершает работу.

Если пользователь выбрал корректное имя файла, то производится формирование отчета в формате HTML с использованием класса `StringBuilder`.

Отчет формируется в виде HTML-таблицы. После формирования отчета с помощью класса `StringBuilder`, отчет сохраняется в текстовый файл с помощью метода «`File.AppendAllText`»:

```
File.AppendAllText(ReportFileName, b.ToString());
```

Первым параметром метода является путь и имя файла, указанные в классе `SaveFileDialog`. Вторым параметром метода является отчет, сформированный с помощью класса `StringBuilder` и преобразуемый к типу `String`.

Далее с помощью класса `MessageBox` выводится сообщение пользователю об имени файла, содержащего отчет.

Отображение файла отчета в браузере представлено на рис. 46.

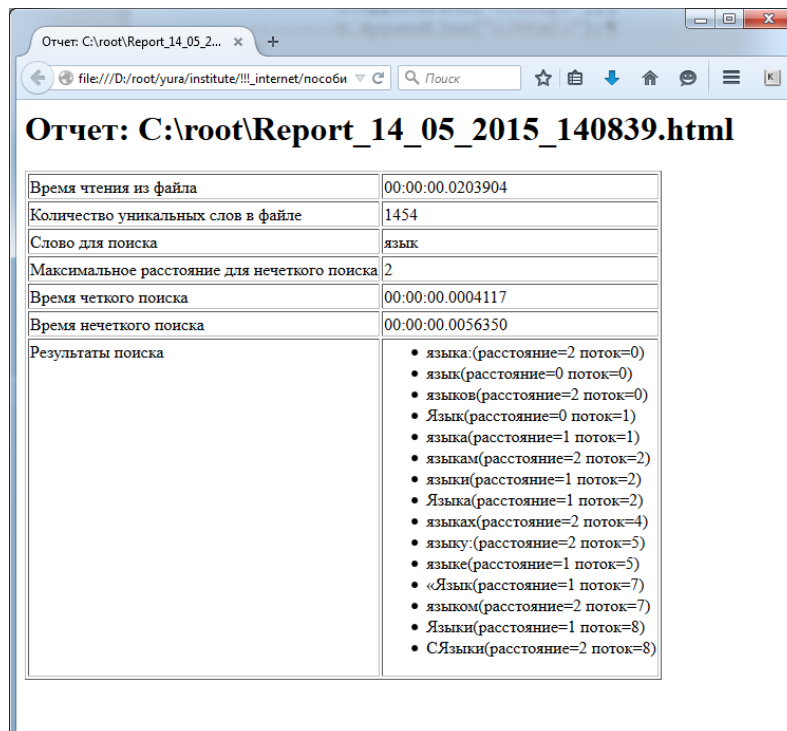


Рис. 46. Отображение файла отчета в браузере.

Текст сформированного файла «Report\_14\_05\_2015\_140839.html»:

```

<html>
<head>
<meta http-equiv='Content-Type' content='text/html; charset=UTF-8' />
<title>Отчет: C:\root\Report_14_05_2015_140839.html</title>
</head>
<body>
<h1>Отчет: C:\root\Report_14_05_2015_140839.html</h1>
<table border='1'>
<tr>
<td>Время чтения из файла</td>
<td>00:00:00.0203904</td>
</tr>
<tr>
<td>Количество уникальных слов в файле</td>
<td>1454</td>
</tr>
<tr>
<td>Слово для поиска</td>
<td>язык</td>
</tr>
<tr>
<td>Максимальное расстояние для нечеткого поиска</td>
<td>2</td>
</tr>
<tr>
<td>Время четкого поиска</td>
<td>00:00:00.0004117</td>
</tr>
<tr>
<td>Время нечеткого поиска</td>
<td>00:00:00.0056350</td>
</tr>
<tr valign='top'>
<td>Результаты поиска</td>
<td>
<ul>
<li>языка:(расстояние=2 поток=0)</li>
<li>язык(расстояние=0 поток=0)</li>
<li>языков(расстояние=2 поток=0)</li>
<li>Язык(расстояние=0 поток=1)</li>
<li>языка(расстояние=1 поток=1)</li>
<li>языкам(расстояние=2 поток=2)</li>
<li>языки(расстояние=1 поток=2)</li>
<li>Языка(расстояние=1 поток=2)</li>
<li>языках(расстояние=2 поток=4)</li>
<li>языку:(расстояние=2 поток=5)</li>
<li>языке(расстояние=1 поток=5)</li>
<li>«Язык(расстояние=1 поток=7)</li>

```



```
<li>языком(расстояние=2 поток=7)</li>
<li>Языки(расстояние=1 поток=8)</li>
<li>СЯзыки(расстояние=2 поток=8)</li>
</ul>
</td>
</tr>
</table>
</body>
</html>
```

## Контрольные вопросы

### *К разделу 2*

1. Что такое MSIL?
2. Что такое CLR?
3. Для чего используется сборщик мусора?
4. Опишите процесс компиляции, компоновки и исполнения программы для .NET- платформы.
5. В чем различие между процессами компиляции и компоновки для языков без использования среды исполнения и для платформы .NET?
6. Что такое JIT-компиляция?
7. Почему машинный код, сформированный JIT-компилятором, может в некоторых случаях выполняться быстрее, чем машинный код, сформированный обычным компилятором?

### *К разделу 3*

1. Что такое CTS?
2. Что такое типы-значения?
3. Что такое ссылочные типы?
4. Какие целочисленные типы данных существуют в C#?
5. Какие типы данных с плавающей точкой существуют в C#?
6. Какие символьные и строковые типы данных существуют в C#?

7. Что такое тип object?
8. Какими способами можно преобразовать значение строкового типа в значение числового типа?
9. Как объявляются одномерные массивы в C#?
10. Как объявляются прямоугольные и зубчатые многомерные массивы в C#? В чем разница между ними?
11. Какие средства консольного ввода/вывода существуют в C#?
12. Как работает форматированный вывод в консоль?
13. Что такое «пространства имен» и «сборки», как они соотносятся друг с другом?
14. Как задаются и обрабатываются аргументы командной строки в консольном приложении?
15. Какие условные операторы существуют в C#?
16. Какие операторы цикла существуют в C#?
17. Как работает цикл foreach?
18. Как работает механизм обработки исключений в C#?
19. В чем особенность порядка расположения блоков catch в операторе обработки исключений?
20. Как работают ref-параметры в C#?
21. Как работают out-параметры в C#?
22. Как передать в метод переменное количество параметров?
23. В чем разница между операторами return и yield return?
24. Как используются XML-комментарии в C#?
25. Как используются директивы препроцессора в C#?

#### ***К разделу 4***

1. Как объявить конструктор класса в C#?
2. Как объявить метод класса в C#?
3. Какие области видимости существуют в 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 C#?
11. Как объявить статические элементы класса в C#?
12. Как реализуется наследование класса от класса?
13. Как из конструктора класса вызвать конструктор базового класса?
14. Как из конструктора класса вызвать другой конструктор текущего класса?
15. Как переопределить виртуальный метод?
16. В чем разница между ключевыми словами «override» и «new» при переопределении виртуального метода?
17. В чем сходства и различия между виртуальными и абстрактными методами?
18. Что такое абстрактный класс?
19. Как в Visual Studio сгенерировать заглушки методов при наследовании от абстрактного класса?
20. Что такое интерфейсы и для чего они используются в C#?
21. В чем сходства и различия между интерфейсами и абстрактными классами?
22. Можно ли в C# унаследовать класс от нескольких классов? От нескольких интерфейсов? Почему?
23. Как реализуется наследование класса от класса и интерфейсов?
24. Являются ли методы, унаследованные от интерфейсов, виртуальными?

25. Как в Visual Studio сгенерировать заглушки методов при наследовании от интерфейса?
26. В чем разница между «реализацией интерфейса» и «явной реализацией интерфейса»?
27. Что такое методы расширения и как они реализуются в C#?
28. Что такое частичные классы и как они реализуются в C#?
29. Как создать диаграмму классов в Visual Studio?

### ***К разделу 5***

1. Что такое перечисление?
2. Как получить значение перечисления на основе строки?
3. Как реализуется перегрузка операторов в C#?
4. Что такое индексатор?
5. Как создать обобщенный класс в C#?
6. Как создать обобщенный метод в C#?
7. Что такое делегат?
8. Что такое неявная типизация и как она используется для делегатов?
9. Как передать в метод параметр типа делегат?
10. Что такое лямбда-выражения и как они используются?
11. Как используется обобщенный делегат Func?
12. Как используется обобщенный делегат Action?
13. Что такое групповой делегат?
14. Как реализуется механизм событий в C#?

### ***К разделу 6***

1. В чем сходство и различие между обобщенными и необобщенными коллекциями?
2. Как осуществляется работа с обобщенным списком?

3. Как осуществляется работа с необобщенным списком?
4. Как осуществляется работа с обобщенным стеком?
5. Как осуществляется работа с обобщенной очередью?
6. Как осуществляется работа с обобщенным словарем?
7. Как осуществляется работа с кортежем?
8. Как осуществляется сортировка коллекций?
9. Как используется интерфейс `IComparable` при сортировке коллекций?
10. Как можно реализовать класс разреженной матрицы на основе класса словаря?
11. Как можно реализовать классы списка и стека без использования стандартных коллекций?

### ***К разделу 7***

1. Как получить список файлов для заданного каталога?
2. Как получить список подкаталогов для заданного каталога?
3. Как осуществить чтение текстового файла в виде единой строки?
4. Как осуществить запись текстового файла в виде единой строки?
5. Как осуществить чтение текстового файла в виде массива строк?
6. Как осуществить запись текстового файла в виде массива строк?
7. Как осуществить сериализацию/десериализацию объекта в бинарный файл?
8. Как осуществить сериализацию/десериализацию объекта в XML-файл?

### ***К разделу 8***

1. Что такое рефлексия?

2. Как реализуется работа со сборками?
3. Как реализуется работа с типами данных?
4. Как реализуются динамические действия с объектами классов?
5. Как реализуется работа с атрибутами?
6. Как можно просмотреть содержимое откомпилированной сборки?

### ***К разделу 9***

1. Как реализовать запуск параллельных потоков на основе класса Thread?
2. Как реализовать запуск параллельных потоков на основе класса Task?
3. Как получить результат выполнения потока с использованием класса Task?
4. Как реализовать ожидания завершения массива потоков с использованием класса Task?
5. Как реализовать измерение времени выполнения программы с использованием класса Stopwatch?
6. Как используются конструкции async и await?

### ***К разделу 10***

1. Что такое расстояние Левенштейна?
2. Что такое расстояние Дамерау-Левенштейна?
3. Объясните алгоритм Вагнера-Фишера вычисления расстояния Дамерау-Левенштейна.

### ***К разделу 11***

1. Как создать приложение Windows Forms?
2. Как используется элемент Button?

3. Как используется элемент TextBox?
4. Как используется элемент Label?
5. Как используется элемент Timer?
6. Как задать форму, которая открывается при запуске приложения?
7. Как открыть модальное окно?
8. Как открыть немодальное окно?
9. Как закрыть окно и приложение?
10. Для чего используются события FormClosed и FormClosing?

### ***К разделу 12***

1. Как используется класс OpenFileDialog?
2. Как используется класс SaveFileDialog?
3. Как используется класс MessageBox?
4. Как используется элемент ListBox?
5. Как сформировать отчет с использованием класса StringBuilder и сохранить его в виде текстового файла?

## Заключение

В учебном пособии были рассмотрены следующие вопросы:

- краткая характеристика среды исполнения .NET;
- базовые конструкции языка программирования C#;
- основы объектно-ориентированного программирования с использованием C#;
- работа с коллекциями;
- основы использования файловой системы;
- рефлексия;
- основы параллельной обработки данных;
- основы технологии Windows Forms.

Учебное пособие в целом соответствует содержанию дисциплины «Базовые компоненты интернет-технологий», которая читается на кафедре «Системы обработки информации и управления» в третьем семестре.

Полученные знания, умения и владения предполагается использовать в рамках дисциплины «Разработка интернет-приложений», которая включает изучение таких технологий как LINQ, Entity Framework, ASP.NET MVC. Поэтому знание языка «C#» является необходимым требованием для изучения данной дисциплины.



## Источники

1. Нейгел К., Ивѐн Б., Глинн Д., Уотсон К. С# 5.0 и платформа .NET 4.5 для профессионалов. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2014. – 1440 с.
2. Шилдт Г. С# 4.0: полное руководство. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2013. – 1056 с.
3. Павловская Т.А. С#. Программирование на языке высокого уровня. Учебник для вузов. – СПб.: Питер, 2015. – 432 с.
4. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.