

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
"САМАРСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ"

А.А. Коробецкая

РАЗРАБОТКА
ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Учебное пособие

Самара
Издательство
Самарского государственного экономического университета
2019

УДК 004.4(075)
ББК 3973.2я7
К68

Рецензенты: *В.М. Дуплякин*, Заслуженный деятель науки и техники РФ, доктор технических наук, профессор кафедры экономики Самарского национального исследовательского университета имени С.П. Королева;

О.П. Солдатова, кандидат технических наук, доцент кафедры информационных систем и технологий Самарского национального исследовательского университета имени академика С.П. Королева

Издается по решению
редакционно-издательского совета университета

Коробецкая, Анастасия Александровна.

К68 Разработка программных приложений [Электронный ресурс] : учеб. пособие / А.А. Коробецкая. - Самара : Изд-во Самар. гос. экон. ун-та, 2019. - 1 электрон. опт. диск. - Систем. требования: процессор Intel с тактовой частотой 1,3 ГГц и выше ; 256 Мб ОЗУ и более ; MS Windows XP/Vista/7/10 ; Adobe Reader ; разрешение экрана 1024×768 ; привод CD-ROM. - Загл. с титул. экрана. - № госрегистрации: 0322000580.
ISBN 978-5-94622-983-8

Учебное пособие содержит теоретические сведения и практические примеры по разработке программных приложений на языке C# в среде Visual Studio. Рассматриваются базис синтаксиса языка C#, основные понятия и принципы объектно-ориентированного программирования, создание приложений с текстовым и графическим пользовательским интерфейсом.

Предназначено для студентов направления подготовки 09.03.03 "Прикладная информатика" и является формой текущего контроля сформированности профессиональных компетенций (ПК-2, ПК-7, ПК-8, ПК-9).

УДК 004.4(075)
ББК 3973.2я7

ISBN 978-5-94622-983-8

© ФГБОУ ВО "Самарский государственный
экономический университет", 2019
© Коробецкая А.А., 2019

ОГЛАВЛЕНИЕ

Введение	4
1. Основы языка C#. Консольное приложение. Ветвление.....	6
2. Основы языка C#. Строки и символы. Циклы. Массивы	53
3. ООП. Классы, атрибуты, методы. Списки	88
4. Принципы ООП	137
5. Создание приложений с графическим пользовательским интерфейсом. Windows Forms	176
Заключение	203
Рекомендуемая литература.....	205

ВВЕДЕНИЕ

Целью разработки является формирование профессиональных навыков и компетенций, связанных с разработкой программных приложений на объектно-ориентированном языке C# в среде Visual Studio.

На данный момент язык C# является одним из наиболее популярных языков программирования в мире, применяющихся в разработке настольных и мобильных приложений. C# отличается довольно простым, но в то же время богатым синтаксисом, вобравшим в себя преимущества языков C, C++, Java, Pascal, и относится к объектно-ориентированным языкам, которые имеют строгую типизацию.

Наиболее часто в качестве разработки на языке C# используется среда Visual Studio, которая отличается удобным и понятным пользовательским интерфейсом, интегрированным отладчиком, встроенной интеллектуальной подсказкой IntelliSense и многими другими инструментами, упрощающими создание прикладных программ. Несомненным достоинством является наличие бесплатной версии Visual Studio Community, включающей весь необходимый набор инструментов для начинающего и даже для профессионального разработчика.

В пособии на практических примерах рассматриваются основы синтаксиса языка C#, основные понятия и принципы объектно-ориентированной разработки приложений, а также создание программных приложений с текстовым и графическим пользовательским интерфейсом.

Материал излагается таким образом, чтобы теоретические сведения перемежались с практической разработкой приложений. В примерах демонстрируются и сравниваются различные подходы и возможные варианты решения задач.

Предполагается, что учащиеся уже знакомы с основами программирования на одном из высокоуровневых языков (Basic, Pascal, Python и др.). Тем не менее, его можно использовать и для первоначального изучения программирования, так как вместе с основами языка C# рассматриваются базовые понятия и алгоритмы программирования.

В дополнение к изложенному материалу рекомендуется изучение официальных источников сети Microsoft Developer Network (MSDN), а

также ознакомление с требованиями и рекомендациями, изложенными в следующих источниках.

1. Руководство по программированию на C# // Microsoft. URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/index>.

2. Соглашения о написании кода на C# // Microsoft. URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>.

3. Рекомендации по написанию кода на C# // Aviva Solutions. URL: <https://csharpcodingguidelines.com/> (перевод на русский язык <https://habr.com/post/272053/>).

4. Документирование кода с помощью XML-комментариев // Microsoft. URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/codedoc>.

После изучения представленного материала студент сможет разрабатывать простые программные приложения с учетом принципов объектно-ориентированного подхода, с текстовым или графическим пользовательским интерфейсом.

1. ОСНОВЫ ЯЗЫКА C#. КОНСОЛЬНОЕ ПРИЛОЖЕНИЕ. ВЕТВЛЕНИЕ

О языке C#

Язык C# (читается "си-шарп", иногда пишут C Sharp, CS) - один из самых популярных языков программирования в мире.

В рейтинге IEEE Spectrum (читается "ай-трипл-и спектрум") за 2018 г. он занимает пятую строчку (рис. 1.1). Как отмечено в рейтинге, он применяется в веб-разработке (backend), разработке мобильных приложений и приложений для настольных компьютеров.

C# был разработан в 1998-2001 гг. фирмой Microsoft как язык разработки приложений для платформы Microsoft .NET Framework. Соответственно, в первую очередь, он ориентирован на разработку под Windows, хотя и не ограничен этим.

Особенности языка C#:

- имеет C-подобный синтаксис, больше всего похож на Java;
- является объектно-ориентированным (все, что есть в языке, реализовано через классы и объекты);
- строго типизирован;
- неразрывно связан с платформой .NET, написанные на нем программы требуют ее установки;
- программы выполняются в среде CLR через JIT-компилятор, что обеспечивает кроссплатформенность;
- основная среда разработки - Microsoft Visual Studio.

Установка VisualStudio

Среда разработки Microsoft VisualStudio поддерживает множество языков программирования, язык C# устанавливается по умолчанию.

Существует несколько версий VisualStudio. В данном пособии рассматривается **Visual Studio Community 2015**. Community - бесплатная версия, необходимо только иметь учетную запись Microsoft.























Language Rank	Types	Spectrum Ranking
1. Python	  	100.0
2. C++	  	99.7
3. Java	  	97.5
4. C	  	96.7
5. C#	  	89.4
6. PHP		84.9
7. R		82.9
8. JavaScript	 	82.6
9. Go	 	76.4
10. Assembly		74.1

Рис. 1.1. Рейтинг языков программирования IEEE-2018*

* URL: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018>

Скачать пакет установки можно на сайте <https://visualstudio.microsoft.com/>.

На данный момент последней версией является 2017, по умолчанию будет предложена именно она. Однако совместимость проектов разных версий достаточно низкая: вы сможете просматривать текст программы из версии 2017 в 2015, но проект для запуска придется создать заново.

Чтобы скачать и установить более старую версию:

1) перейдите по ссылке <https://visualstudio.microsoft.com/ru/vs/older-downloads/>;

2) внизу страницы разверните "2015" - "Скачать";

3) войдите под своей учетной записью Microsoft или зарегистрируйтесь;

4) подтвердите участие в *VisualStudio Dev Essentials* (это бесплатный пакет разработчика под VisualStudio);

5) на вкладке Downloads найдите Visual Studio Community 2015 с последним обновлением, выберите язык Russian (если вы хорошо знаете английский, то лучше оставьте English), нажмите Download;

6) запустите файл установки. Потребуется минимум 6 Гб свободного места. Настройки можно оставить по умолчанию;

7) при первом запуске необходимо указать данные своей учетной записи, на нее будет оформлена бесплатная лицензия.

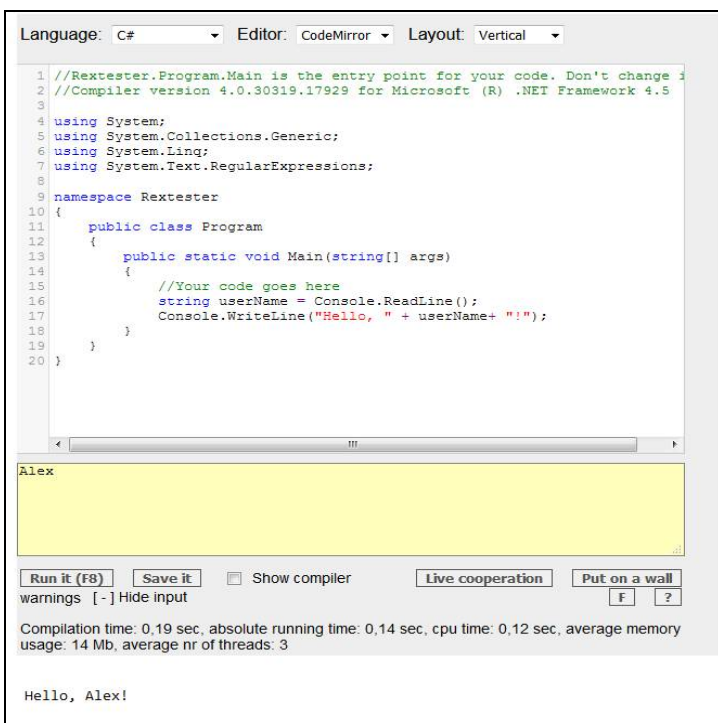


Рис. 1.2. Интерфейс онлайн-компилятора RexTester для языка C#

Другие способы запуска программ

Фактически для разработки небольших учебных консольных приложений VisualStudio не требуется. Нужен лишь компилятор C#, который позволит запустить программу.

Существует множество онлайн-компиляторов под все популярные языки программирования. Один из наиболее популярных - RexTester.Com.

Примечание. Если в браузере включен автоматический перевод на русский язык, обязательно отключите его - это нарушает работоспособность кода.

RexTester позволяет писать и запускать консольные приложения. Для ввода исходных данных нажмите Show input в нижней части окна. Результат работы программы будет выводиться в самом низу страницы.

Зарегистрировавшись, можно сохранять свои программы на сервере, а также совместно просматривать и редактировать код с другими участниками.

На скриншоте на рис. 1.2 показан запуск первого примера из данного пособия в RexTester.

О консольных приложениях

Простейший вид приложений - это приложения с текстовым пользовательским интерфейсом (Text User Interface, TUI), или консольные приложения, или приложения в режиме командной строки. В них пользователь вводит текстовые команды и данные последовательно, по строкам.

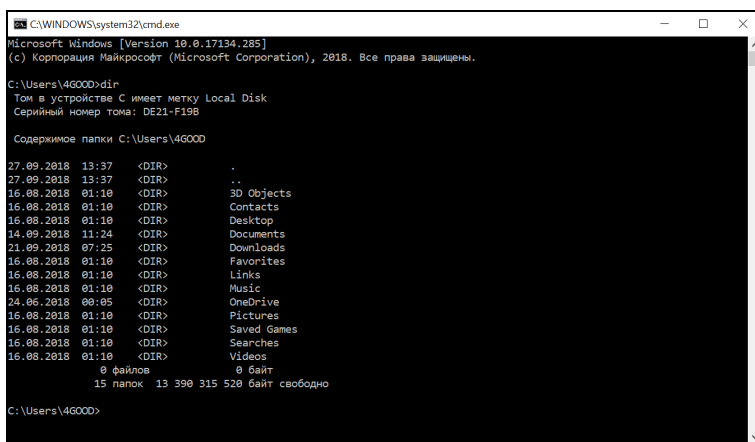
На рис. 1.3 представлена командная строка Windows, в которую была введена команда `dir` для просмотра содержимого текущей директории. В меню "Пуск" соответствующее приложение находится в категории "Служебные" - "Командная строка". Либо можно запустить через окно "Выполнить" (Win+R) - `cmd`.

Хотя может показаться, что консольные приложения давно ушли в прошлое, на самом деле их очень много. Фактически сюда попадают все приложения без графического оконного интерфейса - различные службы, серверы, утилиты.

Зачастую у прикладных программ есть два варианта: графический для рядовых пользователей и консольный для профессионалов, с поддержкой дополнительных настроек и массовой обработки файлов.

Примеры: компиляторы C#, Python и других языков, архиватор 7-Zip, конвертер изображений ImageMagick, утилиты диагностики дисков и оперативной памяти Windows.

Консольные команды можно сохранять в виде CMD или BAT-файлов для многократного запуска и выполнения, чем часто пользуются системные администраторы.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.285]
(c) Корпорация Майкрософт (Microsoft Corporation), 2018. Все права защищены.

C:\Users\4600D>dir
Том в устройстве C имеет метку Local Disk
Серийный номер тома: DE21-F19B

Содержимое папки C:\Users\4600D

27.09.2018 13:37 <DIR>          .
27.09.2018 13:37 <DIR>          ..
16.08.2018 01:10 <DIR>          3D Objects
16.08.2018 01:10 <DIR>          Contacts
16.08.2018 01:10 <DIR>          Desktop
14.09.2018 11:24 <DIR>          Documents
21.09.2018 07:25 <DIR>          Downloads
16.08.2018 01:10 <DIR>          Favorites
16.08.2018 01:10 <DIR>          Links
16.08.2018 01:10 <DIR>          Music
24.05.2018 00:05 <DIR>          OneDrive
16.08.2018 01:10 <DIR>          Pictures
16.08.2018 01:10 <DIR>          Saved Games
16.08.2018 01:10 <DIR>          Searches
16.08.2018 01:10 <DIR>          Videos
           0 файлов          0 байт
           15 папок 13 390 315 520 байт свободно

C:\Users\4600D>
```

Рис. 1.3. Командная строка Windows

Пример 1. Приветствуем пользователя

Задание

Написать консольное приложение, которое запрашивает имя пользователя и выводит приветствие с обращением к пользователю по этому имени.

Если имя пользователя передано в качестве параметра командной строки, то не запрашивать его, а сразу вывести приветствие.

Указания к выполнению

Создание консольного приложения

Начнем с классического примера "Hello, world!", а затем добавим в него запрос имени пользователя.

Запустите среду разработки VisualStudio. Создайте новый проект - консольное приложение: "Файл" (File) → "Создать проект..." (Create Project...) (Ctrl+Shift+N) → в открывшемся окне в категории Visual C# выбрать "Консольное приложение" (Console Application), как показано на рис. 1.4. Задайте имя приложения "HelloUser" (без пробелов) вместо "ConsoleApplication1". Выберите папку, в которой сохранится ваше приложение.

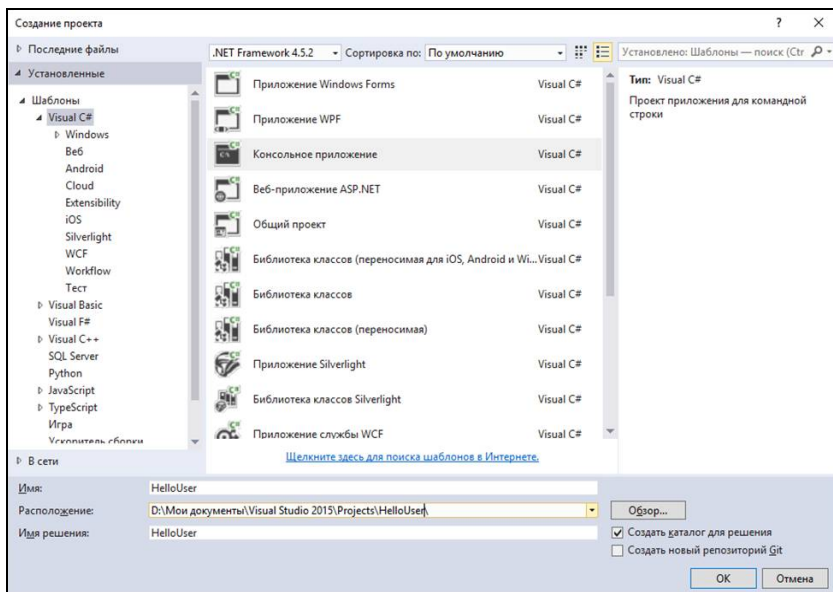


Рис. 1.4. Создание консольного приложения "HelloUser"

? Где сохраняются приложения по умолчанию?

В результате вы увидите каркас программного кода и различные окна, позволяющие управлять программой (рис. 1.5).

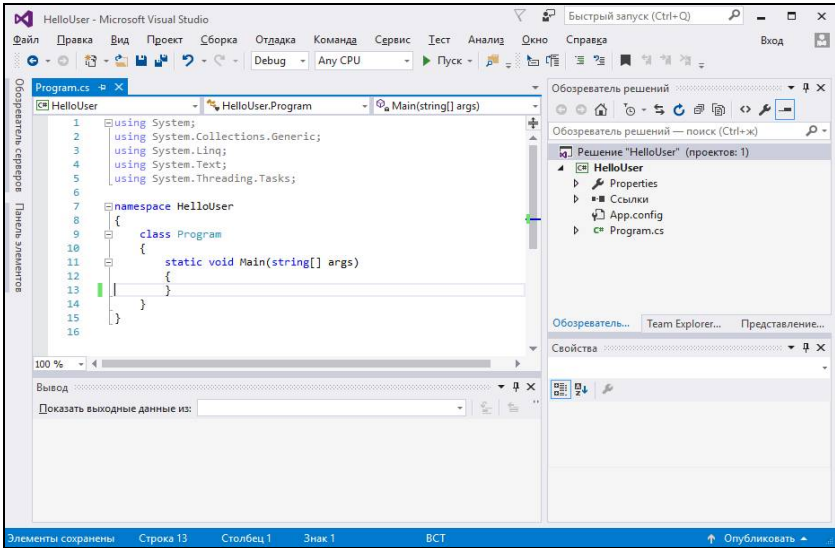


Рис. 1.5. Проект "HelloUser" в окне VisualStudio

Если вы ошиблись в имени проекта или забыли изменить стандартное название, то его можно переименовать в "Обозревателе решений" (Project Explorer), который по умолчанию находится в правой верхней части окна VisualStudio (рис. 1.6).

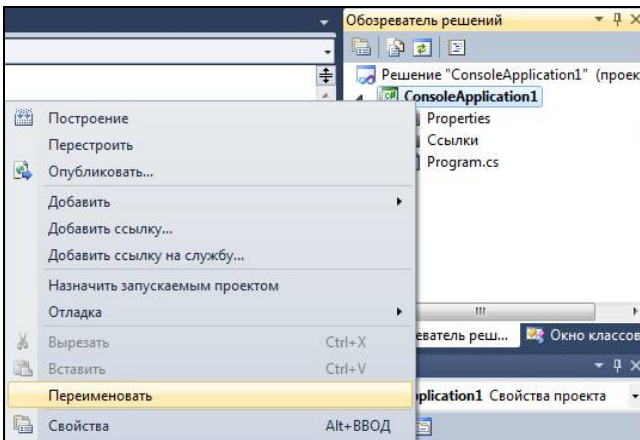
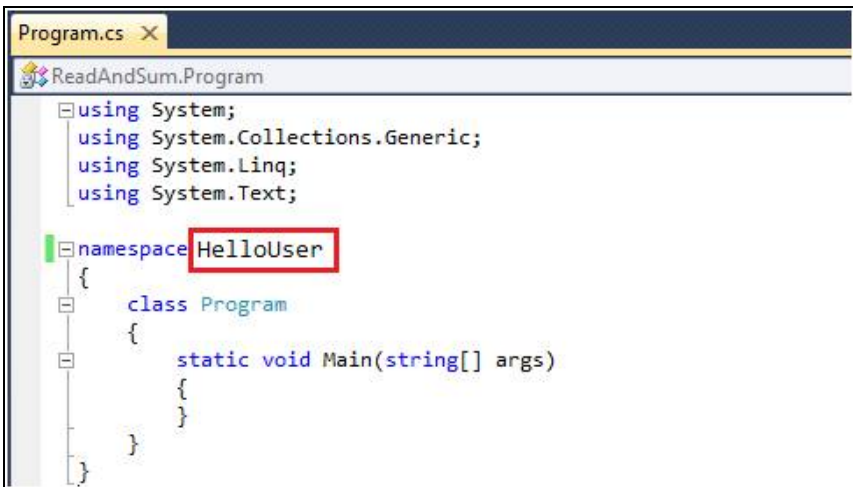


Рис. 1.6. "Обозреватель решений". Изменение имени проекта

Кроме того, нужно изменить название пространства имен (namespace) в исходном коде, как показано на рис. 1.7.



```
Program.cs X
ReadAndSum.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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

Рис. 1.7. Пространство имен проекта

Структура программы на C#

Среда VisualStudio автоматически генерирует каркас программы при создании нового проекта. Рассмотрим его немного подробнее.

using - ключевое слово для ссылки на внешние модули, в которых содержится уже готовый код, используемый в данном приложении. По умолчанию программа ссылается на системный модуль System, который содержит базовые возможности языка (ввод-вывод данных, математические функции, работа со строками и т.д.).

namespace - пространство имен. В пределах одного пространства имени объектов должны быть уникальными. В простых приложениях обычно используется только одно пространство имен, совпадающее с именем приложения. В сложных приложениях, разрабатываемых несколькими программистами, пространств имен может быть несколько десятков. Если пространство имен одно, то допускается его не указывать.

Фигурные скобки {} - ограничивают блок программного кода, который рассматривается как единая часть программы. Внутри скобок текст программы пишется с отступом в начале строки, общепринято равным четырем пробелам или одному символу табуляции.

class - имя класса. C# - объектно-ориентированный язык, абсолютно все в программе представляется в виде объектов и классов. По-

дробнее понятия объектно-ориентированного программирования мы рассмотрим позже, а пока запомните это как факт: ваша программа является классом, который размещается в системной памяти.

По умолчанию класс носит имя **Program**, но его можно переименовать.

Внутри класса **Program** объявлен метод **Main** с ключевыми словами **static** и **void** (их смысл также рассмотрим позже). Метод **Main** переименовывать нельзя - именно по этому имени в языке C# определяется, с чего начинать работу программы.

В круглых скобках указан аргумент **string[] args** - при запуске программы в него будут помещены параметры командной строки (указываются в командной строке через пробел после имени приложения). Если ваша программа не получает никаких параметров, то это объявление можно пропустить, указав просто пустые скобки `()`.

Таким образом, автоматически сгенерированный каркас можно сократить. Самый короткий допустимый код выглядит следующим образом:
`using System;`

```
class Program
{
    static void Main()
    {
    }
}
```

Программный код, который выполнится при запуске программы, следует размещать внутри фигурных скобок метода **Main**.

```
namespace HelloUser
{
    class Program
    {
        static void Main(string[] args)
        {
            // здесь пишется код
        }
    }
}
```

Комментарии

Две косые черты (два слеша) означают начало **комментария**. Комментарии игнорируются при запуске программы. Они нужны для про-

граммиста, читающего программу, чтобы лучше понять ее смысл. Например:

```
// это однострочный комментарий
```

Такой комментарий называется однострочным: он начинается с // и заканчивается в конце строки. После // принято ставить один пробел.

Если комментарий длинный и требуется его разместить в несколько строк, то используются символы /* (начало комментария) и */ (конец комментария).

Примеры:

```
*/ /* А это многострочный комментарий.
```

```
* Обычно он состоит из нескольких предложений.
```

```
* В начале каждой новой строки принято ставить звездочку,
```

```
* но это необязательно.
```

```
*/
```

```
/* Этот комментарий тоже считается многострочным.
```

```
/*
```

Ввод-вывод данных в консоль

Для взаимодействия с консолью используется **класс Console**. Все действия, которые можно с ней выполнить, - это **статические методы** класса Console.

Методы и свойства класса пишутся через точку после его имени. VisualStudio автоматически предложит список доступных методов и их описание. На рис. 1.8 показана подсказка IntelliSense для класса Console. Сразу можно увидеть настройку цвета фона BackgroundColor, стандартный звук Beep, очистку консоли Clear и др.

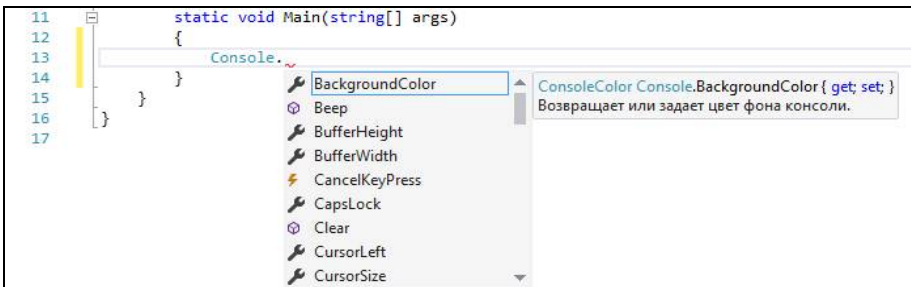


Рис. 1.8. Подсказка свойств и методов класса IntelliSense

Для вывода в консоль используются методы **Write** (напиши) и **WriteLine** (напиши и перейди на новую строку).

Добавьте в исходный код команду, которая выведет в консоль фразу "Hello, World!":

```
7 namespace HelloUser
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            Console.WriteLine("Hello, World!");
14        }
15    }
16 }
```

Текстовые данные (**строки**) пишутся в двойных кавычках и выводятся на экран "как есть". Строка "Hello, World!" передается в метод WriteLine в качестве **аргумента**.

Не забудьте поставить **точку с запятой** в конце команды.

Сохранение приложения

Обратите внимание на желтую полосу слева от текста программы - она отмечает все изменения в коде, которые еще не были сохранены. После сохранения полоса станет зеленой.

```
11 static void Main(string[] args)
12 {
13     Console.WriteLine("Hello, World!");
14 }
```

Для сохранения на панели инструментов VisualStudio есть две кнопки:

- "Сохранить" (Ctrl+S) записывает на диск только текущий файл.

- "Сохранить все" (Ctrl+Shift+S) записывает на диск все файлы текущего проекта, в том числе служебные.

Не забывайте сохранять все файлы проекта. Несохраненные файлы отмечаются звездочкой возле имени.

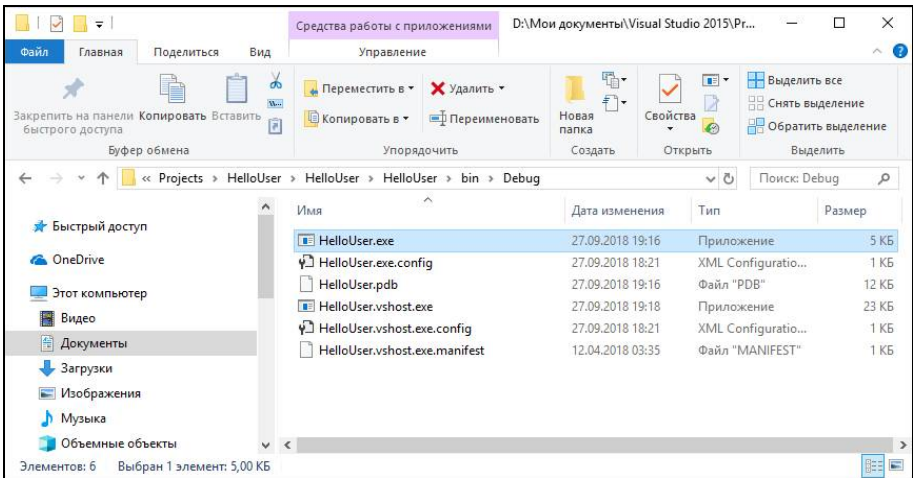


Рис. 1.9. Размещение exe-файла в папке проекта

Запуск приложения

Запустите программу (F5 или кнопка "▶ Пуск" на панели инструментов). Первый запуск может занимать продолжительное время. Затем, скорее всего, вы на секунду увидите окно консоли, которое тут же закроется. Так и должно быть, по завершении работы программы VisualStudio автоматически закрывает консоль.

Запустим нашу программу не через VisualStudio, а просто через exe-файл (он автоматически создается при каждом запуске или по команде "Компилировать" в меню).

Найдите папку с решением, которую вы указали при создании приложения (по умолчанию это %Документы%\Visual Studio 2015\Projects\). В ней зайдите в папку проекта (по умолчанию имя совпадает с решением HelloUser), далее bin\Debug (рис. 1.9).

Откройте командную строку Windows и запустите вашу программу, указав полный путь к exe-файлу, как показано на рис. 1.10. Если путь содержит пробелы, то его следует писать в кавычках.

```
C:\Users\user>"D:\Мои документы\Visual Studio 2015\Projects\HelloUser\HelloUser\HelloUser\bin\Debug\HelloUser.exe"
Hello, world!
```

Рис. 1.10. Запуск программы "HelloUser" в командной строке Windows

Здесь мы увидели, что программа действительно вывела требуемый текст.

Вернемся в VisualStudio. Все-таки обычно мы будем запускать программы в ней, а не через консоль Windows.

Добавьте в вашу программу еще одну строку с командой `Console.ReadKey()`. Эта команда считывает нажатие одной любой кнопки на клавиатуре. Поскольку результат никуда не сохраняется, приложение просто будет ждать, пока пользователь не нажмет что-нибудь.

```
11 static void Main(string[] args)
12 {
13     Console.WriteLine("Hello, world!");
14     Console.ReadKey();
15 }
```

Убедитесь, что приложение запускается из VisualStudio и ждет реакции пользователя.

Переменные

Теперь добавим ввод имени пользователя командой `Console.ReadLine()` - она считывает все, что введет пользователь, до нажатия Enter.

Имя пользователя нужно сохранить в **переменную**. Объявите новую переменную с именем `userName` и типом данных - строка текста (`string`). Порядок **объявления переменной**:

тип данных имя_переменной;

Объявлять переменные можно в любой части программы, главное - до первого использования.

В одной строке с объявлением можно выполнить **инициализацию** переменной, **присвоив** ей какое-то значение:

тип данных имя_переменной = значение;

Таким образом, присваивание в C#, как и во всех C-подобных языках, записывается символом `=`.

```
11 static void Main(string[] args)
12 {
13     string userName = Console.ReadLine();
14     Console.WriteLine("Hello, world!");
15     Console.ReadKey();
16 }
```

Обратите внимание, как записано имя переменной: без пробелов, первое слово `user` с маленькой буквы, второе `Name` с заглавной. И любой человек, немного знающий английский, сразу поймет смысл этой переменной. Такая запись называется **camelStyle** (верблюжий, или горбчатый, стиль).

C# чувствителен к регистру, поэтому `username` - это другая переменная, нельзя `Console` писать с маленькой буквы, а `string` - с заглавной.

Выведем вместо `"World"` введенное имя пользователя. Для этого нужно сцепить текст `"Hello, "`, имя пользователя и восклицательный знак `"!"`.

Конкатенация (сцепка) **строк** осуществляется оператором `+`. Обратите внимание на расстановку пробелов.

```
11     static void Main(string[] args)
12     {
13         string userName = Console.ReadLine();
14         Console.WriteLine("Hello, " + userName + "!");
15         Console.ReadKey();
16     }
```

Запустите приложение и убедитесь в правильности его работы.

Консольное приложение с параметром

Добавим возможность указать имя пользователя сразу при запуске приложения в консоли в качестве параметра после названия приложения. Тогда имя пользователя следует брать из массива `args`.

Закомментируйте строку, в которой имя пользователя вводится из консоли, а вместо нее добавьте присваивание нулевого элемента из массива `args`.

```
13     //string userName = Console.ReadLine();
14     string userName = args[0];
15     Console.WriteLine("Hello, " + userName + "!");
```

Теперь при запуске приложения нам нужно передать ему этот параметр. В окне "Обозревателя решений" кликните по своему проекту "HelloUser" правой кнопкой и выберите "Свойства". Здесь собраны сведения о вашем приложении и его текущей версии. Перейдите на вкладку "Отладка" и в качестве параметра укажите любое имя пользователя (рис. 1.11).

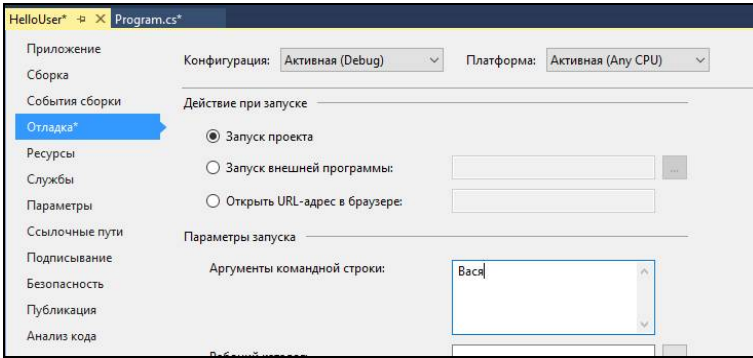


Рис. 1.11. Аргументы командной строки в параметрах проекта

Запустите приложение, и оно, не задавая вопроса пользователю, сразу выведет приветствие с указанным именем (рис. 1.12).

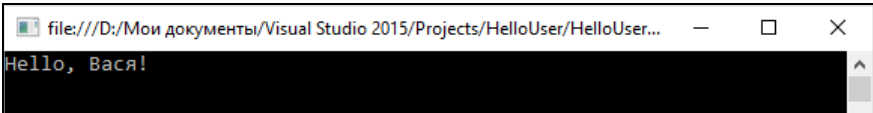


Рис. 1.12. Результат запуска приложения с аргументом

Попробуйте запустить приложение с аргументом в консоли Windows (рис. 1.13).

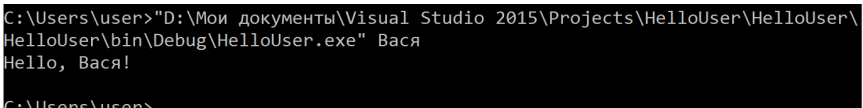


Рис. 1.13. Результат запуска приложения с аргументом в командной строке Windows

Однако при попытке запуска без аргументов приложение выдаст ошибку (рис. 1.14).

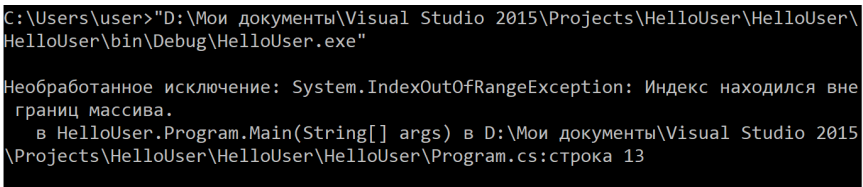


Рис. 1.14. Ошибка при запуске приложения без указания аргумента в командной строке Windows

Проверка условий

Нехорошо, когда программа выдает ошибки сразу после запуска. Добавим проверку, были ли переданы параметры в программу.

Для этого нужен **условный оператор if**. Его синтаксис одинаков во всех С-подобных языках:

```
if (/*условие*/)
{
    //если условие выполняется
}
else
{
    //если условие не выполняется
}
```

Все скобки ставить обязательно. Допустимым является пропуск фигурных скобок для одной команды, но CodeStyle предписывает указывать их всегда.

Операторы сравнения записываются следующим образом:

== равенство (обязательно 2 равно!)
!= неравенство (без пробела между ! и =, ! читается как "не")
> строго больше
< строго меньше
>= больше или равно (синоним не меньше)
<= меньше или равно (синоним не больше)

В нашем примере условие формулируется так: *если* в программу переданы параметры (т.е. их количество больше 0), *то* взять имя пользователя из 0-го параметра, *иначе* запросить имя пользователя в консоли.

Мы не зря сохранили запрос имени пользователя в комментарии - пришло время извлечь его оттуда и поместить в блок **else**.

Количество параметров можно узнать через свойство `args.Length`.

```
11  static void Main(string[] args)
12  {
13      if (args.Length > 0)
14      {
15          string userName = args[0];
16      }
17      else
18      {
19          string userName = Console.ReadLine();
20      }
21      Console.WriteLine("Hello, " + userName + "!");
22      Console.ReadKey();
23  }
```


Однако, как видите, VisualStudio подсвечивает как ошибку `userName` в команде `WriteLine`. Наведите курсор на ошибку, чтобы получить подсказку IntelliSense (рис. 1.15).

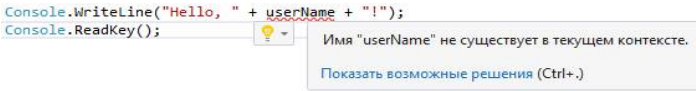


Рис. 1.15. Подсказка IntelliSense при синтаксической ошибке

Ошибка указывает на то, что переменная `userName` не объявлена в текущей части кода (контексте), хотя объявление присутствует в 15 и 19 строках. Дело в том, что **область действия** переменной ограничивается ближайшими фигурными скобками. Поэтому `userName` существует только внутри блоков `if` и `else`, но не во всей остальной программе. Необходимо расширить ее область действия: вынести объявление переменной из `if` в основной код функции `Main`. При этом присваивания остаются там же, где были.

```
13 |         string userName;
14 |         if (args.Length > 0)
15 |         {
16 |             userName = args[0];
17 |         }
18 |         else
19 |         {
20 |             userName = Console.ReadLine();
21 |         }
22 |         Console.WriteLine("Hello, " + userName + "!");
```

Запустите приложение с параметром и без, убедитесь, что оно работает без ошибок в обоих случаях.

Пример 2. Арифметика в консоли

Задание

Написать консольное приложение, которое будет запрашивать два числа, и выполнять с ними все арифметические операции (сложение, вычитание, умножение, деление, деление с остатком).

Проверить работу программы для разных типов данных: `int`, `byte`, `double`, `decimal`.

Указания к выполнению

Создайте новое консольное приложение с именем "ConsoleArithmetic". Сразу добавьте в конец программы

`Console.ReadKey()`, чтобы программа ожидала нажатия любой клавиши, прежде чем закрыться.

Текстовый пользовательский интерфейс

В предыдущем примере мы для простоты пренебрегли важной частью программы: пользовательским интерфейсом. В консольных приложениях он тоже есть, только текстовый, а не графический. Программа должна сообщать пользователю, что она делает, каких ожидает от него действий и в чем заключаются ошибки, если они возникли.

Для начала добавим описание смысла программы и строку, указывающую, что для завершения работы нужно нажать любую клавишу.

```
13 Console.WriteLine("ConsoleArithmetic - все арифметические операции с двумя числами");
14 Console.WriteLine("=====");
15
16 Console.Write("Для завершения работы нажмите любую клавишу...");
17 Console.ReadKey();
```

Убедитесь, что программа запускается без ошибок. Обратите внимание; мы использовали `Write`, а не `WriteLine` перед `ReadKey`, поэтому курсор будет мигать в той же строке после многоточия.

Преобразование строки в число

Нам необходимо, чтобы пользователь ввел в консоль два числа. Базовый целочисленный тип данных - **integer**, пишется сокращенно **int**.

Объявите две переменных типа **int** - `number1` и `number2`, добавьте пояснения для пользователя и считывание значений из консоли.

```
13 Console.WriteLine("ConsoleArithmetic - все арифметические операц
14 Console.WriteLine("=====");
15 Console.WriteLine("Введите два целых числа");
16 int number1 = Console.ReadLine();
17 int number2 = Console.ReadLine();
18
19 Console.Write("Для завершения работы нажмите любую клавишу...");
```

Как видите, не все получилось гладко, IntelliSense указывает ошибку при присваивании: не удастся явно преобразовать тип "string" в "int".

Действительно, в консоль пользователь может ввести любые символы, а из этих символов нам необходимо получить целые числа. Для этого в каждый числовой тип данных "зашит" метод `Parse`.

Напомним, что C# - это объектно-ориентированный язык, и буквально все в нем является **объектами**, в том числе и типы данных. А

значит, после `int` можно поставить точку и посмотреть, что предложит IntelliSense (рис. 1.16).

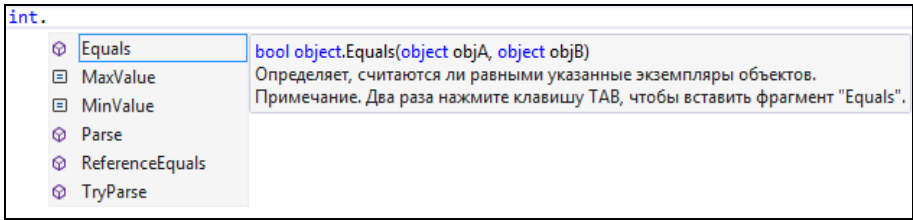


Рис. 1.16. Методы типа данных `int` как класса

Мы видим четыре метода: `Equals`, `Parse`, `ReferenceEquals`, `TryParse`; и два свойства: `MaxValue`, `MinValue`. Обратите внимание, у методов (действий) и свойств (значений) разные пиктограммы.

Нам нужен метод `Parse`. Термин "parse" означает "синтаксический разбор", "анализ". **Парсинг**, в общем смысле, - это разбор текста и извлечение из него полезных данных. Например, компилятор C# парсит ваш программный код, поисковые роботы парсят сайты web-страниц. Мы будем парсить числа из консоли.

```
15 Console.WriteLine( введите два целых числа );
16 int number1 = int.Parse(Console.ReadLine());
17 int number2 = int.Parse(Console.ReadLine());
18
19 Console.WriteLine("Введены числа " + number1 + ", " + number2);
20 Console.WriteLine("Для завершения работы нажмите любую клавишу.");
```

Таким образом, для преобразования текста в число надо использовать метод `Parse` для соответствующего типа данных. Обратное преобразование из числа в текст выполняется автоматически, "на лету".

Для проверки выведем в консоль числа, которые ввел пользователь. Напомним, что оператор `+` для строк означает конкатенацию.

```
15 Console.WriteLine( введите два целых числа );
16 int number1 = int.Parse(Console.ReadLine());
17 int number2 = int.Parse(Console.ReadLine());
18
19 Console.WriteLine("Введены числа " + number1 + ", " + number2);
20 Console.WriteLine("Для завершения работы нажмите любую клавишу.");
```

Проверьте работу программы (рис. 1.17).

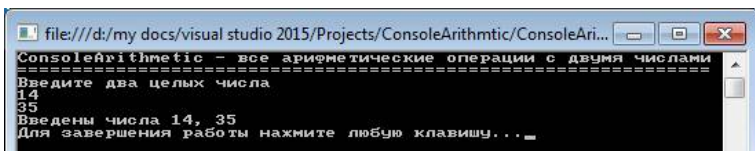


Рис. 1.17. Тестовый запуск программы для ввода-вывода чисел в консоль

Арифметические операции

Добавим в программу вычисления. Всего в C# присутствует пять арифметических операторов:

- 1) сложение + (плюс);
- 2) вычитание - (минус);
- 3) умножение * (звездочка);
- 4) деление / (слеш, прямая косая черта);
- 5) остаток от деления % (символ процента).

Деление для целых чисел выполняется нацело, для дробных - как обычно. Например:

$$23 / 5 = 4$$

$$23 \% 5 = 3$$

$$23.0 / 5 = 4.6$$

Начнем со сложения. Добавим в программу переменную `sum`, которой присвоим результат операции. Выведем получившееся значение в консоль в развернутом виде. Строка, в которой вы просто выводили исходные числа, больше не нужна, можно переписать ее.

```

17         int number2 = int.Parse(Console.ReadLine());
18
19         int sum = number1 + number2;
20         Console.WriteLine(number1 + " + " + number2 + " = " + sum);
21
22         Console.Write("Для завершения работы нажмите любую клавишу... ");

```

Обратите внимание, что в строке 19 + означает сложение, в строке 20 - конкатенацию, а " + " в кавычках - это текст, который выведется в консоль в явном виде.

Протестируйте работу программы (рис. 1.18).

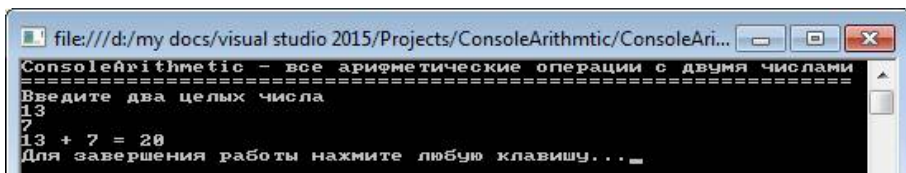


Рис. 1.18. Тестовый запуск программы для сложения целых чисел

По аналогии добавим остальные операторы (**subtraction** - вычитание, **multiplication** - умножение, **division** - деление, **remainder** - остаток).

```
19         int sum = number1 + number2;
20         Console.WriteLine(number1 + " + " + number2 + " = " + sum);
21         int sub = number1 - number2;
22         Console.WriteLine(number1 + " - " + number2 + " = " + sub);
23         int mul = number1 * number2;
24         Console.WriteLine(number1 + " * " + number2 + " = " + mul);
25         int div = number1 / number2;
26         Console.WriteLine(number1 + " / " + number2 + " = " + div);
27         int rem = number1 % number2;
28         Console.WriteLine(number1 + " % " + number2 + " = " + rem);
29     }
```

Результат работы программы для чисел 5 и 11 показан на рис. 1.19.

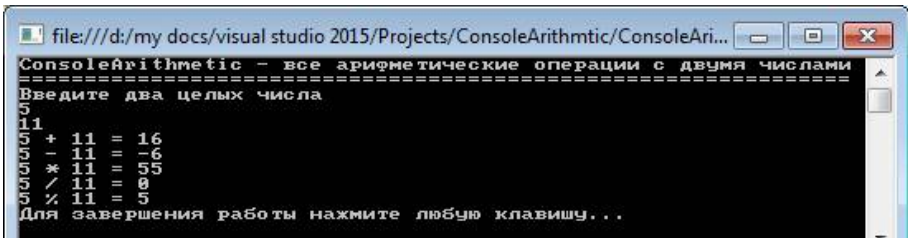


Рис. 1.19. Тестовый запуск программы для сложения целых чисел

Примечание. Можно было обойтись без дополнительных переменных, записав арифметические операции прямо внутрь `WriteLine`.

```
Console.WriteLine(number1 + " - " + number2 + " = " + (number1 - number2));
```

Но такая запись получается слишком сложной для чтения, поэтому рекомендуется ее избегать.

Составной формат строк

Мы использовали конкатенацию для формирования строк в консоли, но, как видите, получается довольно длинная и сложная конструкция с множеством символов + и кавычек, в которых легко запутаться.

Предпочтительным способом является **составной формат строк**: сначала пишем всю строку, а потом перечисляем значения, которые нужно в нее подставить (рис. 1.20).

```
Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);
```



Рис. 1.20. Подстановка значений в строку в составном формате

Тогда команды вывода примут вид:

```
19 int sum = number1 + number2;
20 Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);
21 int sub = number1 - number2;
22 Console.WriteLine("{0} - {1} = {2}", number1, number2, sub);
23 int mul = number1 * number2;
24 Console.WriteLine("{0} * {1} = {2}", number1, number2, mul);
25 int div = number1 / number2;
26 Console.WriteLine("{0} / {1} = {2}", number1, number2, div);
27 int rem = number1 % number2;
28 Console.WriteLine("{0} % {1} = {2}", number1, number2, rem);
~~
```

Кроме того, составной формат позволяет настроить внешний вид выводимых значений, но об этом мы поговорим позже.

Убедитесь, что результат работы остался верным.

Тестирование приложения

Тестирование правильности работы приложения, выявление и устранение ошибок - отдельная большая тема. Существует несколько видов тестирования на разных уровнях и этапах разработки и отдельная профессия - тестировщик.

Тем не менее, любой разработчик должен уметь тестировать правильность своей программы и обнаруживать в ней ошибки. Тестировать программу нужно как можно чаще, так будет проще понять, в какой момент возникла ошибка. Для этого необходимо правильно подобрать **тестовые примеры**. Они должны охватывать различные ситуации, но их должно быть не слишком много.

Выше мы тестировали программу на достаточно простых примерах, которые доказывали правильность ее работы. Но очень важно проверить стабильность работы приложения при неверных и критических значениях (0, отрицательные числа, целые и дробные значения, очень большие и очень маленькие значения, длинный текст, пустая строка).

Лучше всего тестирование оформить в табличной форме, как показано в табл. 1.1.

Таблица 1.1

Тестирование приложения ConsoleArithmetic на типе данных int

№ теста	Входные данные	Ожидаемый результат	Результат программы	Примечание
1	2	3	4	5
1	14 5	$14 + 5 = 19$ $14 - 5 = 9$ $14 \cdot 5 = 70$ $14 / 5 = 2$ $14 \% 5 = 4$	<pre> Введите два целых числа: 14 5 14 + 5 = 19 14 - 5 = 9 14 * 5 = 70 14 / 5 = 2 14 % 5 = 4 Для завершения нажмите Enter </pre>	Первое число больше второго
2	5 9	$5 + 9 = 14$ $5 - 9 = -4$ $5 \cdot 9 = 45$ $5 / 9 = 0$ $5 \% 9 = 5$	<pre> Введите два целых числа: 5 9 5 + 9 = 14 5 - 9 = -4 5 * 9 = 45 5 / 9 = 0 5 % 9 = 5 Для завершения нажмите Enter </pre>	Второе число больше первого
3	10 10	$10 + 10 = 20$ $10 - 10 = 0$ $10 \cdot 10 = 100$ $10 / 10 = 1$ $10 \% 10 = 0$	<pre> Введите два целых числа: 10 10 10 + 10 = 20 10 - 10 = 0 10 * 10 = 100 10 / 10 = 1 10 % 10 = 0 Для завершения нажмите Enter </pre>	Числа равны
4	0 20	$0 + 20 = 20$ $0 - 20 = -20$ $0 \cdot 20 = 0$ $0 / 20 = 0$ $0 \% 20 = 0$	<pre> Введите два целых числа: 0 20 0 + 20 = 20 0 - 20 = -20 0 * 20 = 0 0 / 20 = 0 0 % 20 = 0 Для завершения нажмите Enter </pre>	Первое 0
5	60 0	$60 + 0 = 60$ $60 - 0 = 60$ $60 \cdot 0 = 0$ На 0 делить нельзя	<pre> Введите два целых числа: 60 0 Вылет при делении с ошибкой DivideByZeroException </pre>	Второе 0
6	-37 -12	$-37 + -12 = -49$ $-37 - -12 = -25$ $-37 \cdot -12 = 444$ $-37 / -12 = 3$ $-37 \% -12 = -1$	<pre> Введите два целых числа: -37 -12 -37 + -12 = -49 -37 - -12 = -25 -37 * -12 = 444 -37 / -12 = 3 -37 % -12 = -1 Для завершения нажмите Enter </pre>	Отрицательные
7	1000 123	$1000 + 123 = 1123$ $1000 - 123 = 877$ $1000 \cdot 123 = 123000$ $1000 / 123 = 8$ $1000 \% 123 = 16$	<pre> Введите два целых числа: 1000 123 1000 + 123 = 1123 1000 - 123 = 877 1000 * 123 = 123000 1000 / 123 = 8 1000 % 123 = 16 Для завершения нажмите Enter </pre>	Большие значения

1	2	3	4	5
8	9876543210 123456789	9876543210 + +123456789 = =9999999999 9876543210 - 123456789= = 9753086421 9876543210 · 123456789 = =1219326311126350000 9876543210 / 123456789= = 80 9876543210 % 123456789= = 90	Вылет после ввода первого числа с ошибкой OverflowException	Очень большие значения

Примечание. Используйте Excel для расчета ожидаемого результата.

Пока ограничимся предположением, что наш пользователь следует инструкциям и будет вводить только целые числа, как его и просят.

И без этого мы получили 2 вылета на 8 тестах (25 %). Проанализируем результаты.

Тест № 5 показывает, как должна вести себя программа, когда пользователь пытается делить на 0. Тест предусматривает, что сложение, вычитание и умножение нужно выполнить как обычно, а вместо деления и остатка от деления нужно вывести фразу "на 0 делить нельзя".

Добавим в программу проверку значения number2.

Проверка на равенство записывается двойным символом =. Если number2=0, то выводим сообщение, иначе - выполняем деление.

```

24 | Console.WriteLine("{0} * {1} = {2}", number1, number2, mul);
25 | if (number2 == 0)
26 | {
27 |     Console.WriteLine("на 0 делить нельзя");
28 | }
29 | else
30 | {
31 |     int div = number1 / number2;
32 |     Console.WriteLine("{0} / {1} = {2}", number1, number2, div);
33 |     int rem = number1 % number2;
34 |     Console.WriteLine("{0} % {1} = {2}", number1, number2, rem);
35 | }

```

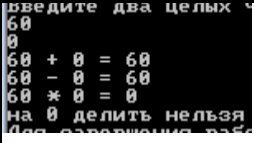
Обратите внимание на **отступы в начале строки**: каждые фигурные скобки добавляют 4 пробела. Чтобы сдвинуть уже набранный текст, выделите его и нажмите Tab.

Чтобы **автоматически отформатировать** весь файл, выберите в меню "Правка" - "Дополнительно" - "Форматировать документ" (или последовательность Ctrl+K, Ctrl+D). Заодно расставятся пробелы в формулах.

Убедимся, что теперь программа проходит этот тест (табл. 1.2).

Таблица 1.2

Тест № 5 после добавления проверки деления на 0

5	60	60 + 0 = 60		Второе 0
	0	60 - 0 = 60		
		60 · 0 = 0 На 0 делить нельзя		

Тест № 8 на самом деле спорный. Возникает вопрос: а могут ли на вход программы попасть такие большие значения? Это зависит от смысла чисел - вряд ли цена на розничный товар или скорость автомобиля могут быть такими большими. Если же большие значения допустимы, то необходимо использовать другой тип данных.

Дело в том, что тип **int** занимает в памяти 4 байта = 32 бита. Из них первый бит отводится под знак числа (0 - положительное, 1 - отрицательное в дополнительном коде), а остальные (31) - под значение. В двоичной системе самое большое число, которое можно записать в 31 бит, равно $2^{31} \approx 2,1$ млрд.

Это число можно увидеть в свойстве **int.MaxValue** (рис. 1.21).

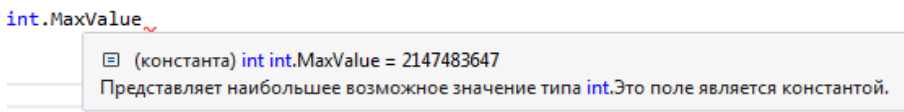


Рис. 1.21. Максимальное значение типа int

? Чему равно **int.MinValue**?

Если тип данных занимает больше места в памяти, то и числа в него можно поместить более длинные. Но все равно будет какой-то предел, допустимый диапазон. Строго говоря, его нужно прописать в условии (техническом задании).

Поскольку мы не знаем экономического смысла двух чисел и диапазоны их значений в условии не заданы, то можно ограничиться типом **int**, а от тестирования таких больших значений отказаться.

Числовые типы данных

Кроме типа `int`, в языке C# предусмотрено еще несколько числовых типов данных. На рис. 1.22 и в табл. 1.3 представлены все числовые типы данных языка C#.

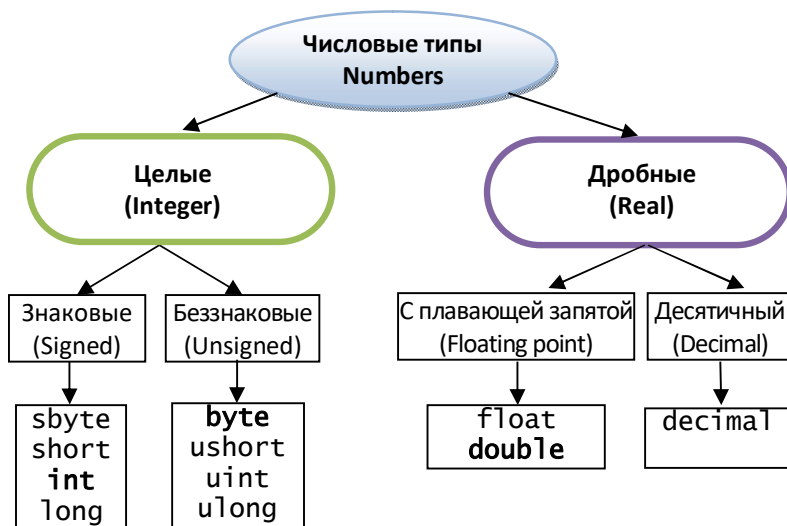


Рис. 1.22. Классификация числовых типов данных

Таблица 1.3

Диапазоны значений числовых типов данных

Категория	Тип	Диапазон значений	Размер (байт)
Целые	sbyte	-128 до 127	1
	byte	0 до $2^8 = 255$	1
	short	-32 768 до 32 767	2
	ushort	0 до $2^{16} = 65\,535$	2
	int	-2^{31} до $2^{31}-1$	4
	uint	0 до $2^{32} = 4\,294\,967\,295$	4
	long	-2^{63} до $2^{63}-1$	8
	ulong	0 до $2^{64}-1 \approx 18,4 \cdot 10^{18}$	8
Дробные	float	$1,5 \cdot 10^{-45}$ до $3,4 \cdot 10^{38}$	4
	double	$5,0 \cdot 10^{-324}$ до $1,7 \cdot 10^{308}$	8
	decimal	$\pm 1,0 \cdot 10^{28}$ до $\pm 7,9 \cdot 10^{28}$	12

Примечание. Строго говоря, decimal - тоже тип с плавающей запятой, но он сконструирован так, чтобы минимизировать ошибки при

округлении десятичных дробей и лучше всего подходит для денежных величин.

Рассмотрим различия в вычислениях между `int`, `byte`, `double` и `decimal`. Тип `int` мы уже разобрали. Посмотрим, что будет для других типов.

Tun byte

Измените типы всех переменных и метода `Parse` на `byte`. Лучше всего воспользоваться поиском и заменой (`Ctrl+H`). Замените все `int` на `byte` (рис. 1.23) с помощью кнопки "Заменить все".

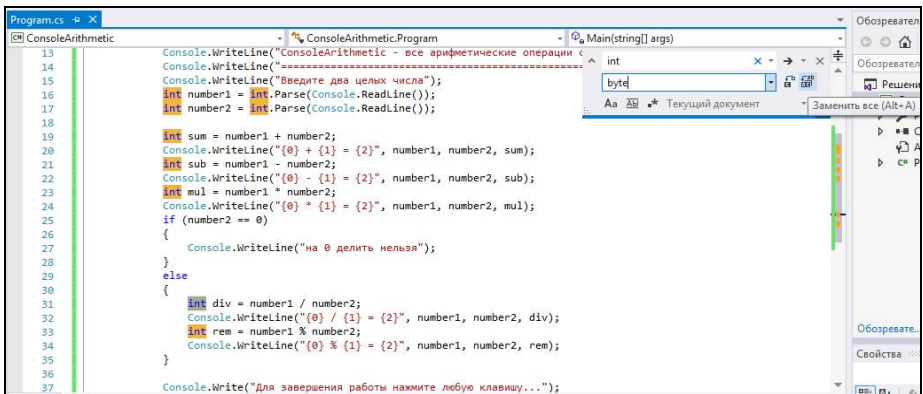


Рис. 1.23. Замена всех `int` на `byte`

Тут же возникла проблема. IntelliSense указывает, что не удастся преобразовать тип `int` в `byte` (рис. 1.24). Но ведь переменные `number1` и `number2` тоже имеют тип `byte`. Откуда же взялся `int`?

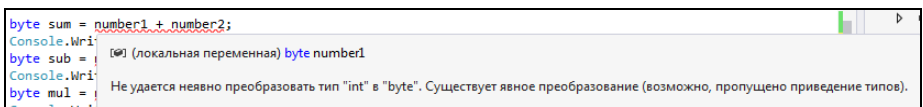


Рис. 1.24. Ошибка при сложении переменных типа `byte`

Дело в том, что `byte` - очень маленький тип данных, и за его границы от 0 до 255 легко выйти при выполнении расчетов. Например, 150 и 200 можно сохранить в переменную типа `byte`, но их сумму - уже нет: $150 + 200 = 350 > 255$. Поэтому по умолчанию результаты всех арифметических операций с участием `byte` преобразуются в тип `int`.

У нас есть два выхода: либо изменить тип переменных `sum`, `sub`, `mul`, `div`, `rem` на `int`, либо выполнить приведение типов, т.е. явно указать, что мы хотим получить ответ типа `byte`.

Приведение типов выполняется с помощью конструкции вида:
(**тип_данных**) выражение

Если выражение длиннее, чем одно значение, то его тоже нужно взять в скобки.

Например, сложение примет вид:

```
18 |
19 | byte sum = (byte)(number1 + number2);
20 | Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);
```

Аналогично измените запись остальных операций.

Протестируем работу приложения (табл. 1.4). Отрицательные числа исключаем, так как они не входят в допустимый диапазон типа `byte`. В качестве больших чисел возьмем 150 и 200.

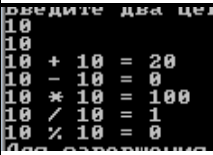
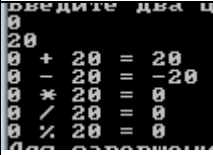
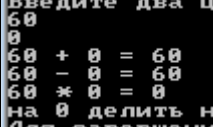
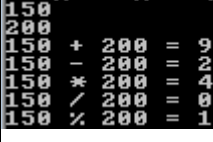
Результат в **тесте № 6** может показаться неожиданным. Но на самом деле все верно. Ведь мы ограничились диапазоном 0-255, а ответ выходит за него.

Разберемся, как работает приведение типов. Для этого числа нужно записать в двоичной системе.

Таблица 1.4

Тестирование приложения ConsoleArithmetic на типе byte

№ теста	Входные данные	Ожидаемый результат	Результат программы	Примечание
1	2	3	4	5
1	14 5	14 + 5 = 19 14 - 5 = 9 14 · 5 = 70 14 / 5 = 2 14 % 5 = 4	Введите два ц 14 5 14 + 5 = 19 14 - 5 = 9 14 * 5 = 70 14 / 5 = 2 14 % 5 = 4 для завершения	Первое число больше второго
2	5 9	5 + 9 = 14 5 - 9 = -4 5 · 9 = 45 5 / 9 = 0 5 % 9 = 5	5 9 5 + 9 = 14 5 - 9 = -4 5 * 9 = 45 5 / 9 = 0 5 % 9 = 5 для завершения	Второе число больше первого

1	2	3	4	5
3	10 10	10 + 10 = 20 10 - 10 = 0 10 · 10 = 100 10 / 10 = 1 10 % 10 = 0		Числа равны
4	0 20	0 + 20 = 20 0 - 20 = -20 0 · 20 = 0 0 / 20 = 0 0 % 20 = 0		Первое 0
5	60 0	60 + 0 = 60 60 - 0 = 60 60 · 0 = 0 на 0 делить нельзя		Второе 0
6	150 200	150 + 200 = 350 150 - 200 = -50 150 · 200 = 30000 150 / 200 = 0 150 % 200 = 150		Большие значения

В типе **int**, который состоит из 4 байт, число 150 будет выглядеть так:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 1 0 1 1 0

А 200 так:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 1 0 0 1 0 0 0

Их сумма 350:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 1 0 1 1 1 1 0

То есть число 350 занимает более 1 байта. При приведении типов в тип **byte** помещаются только младшие 8 бит, а все остальное отбрасывается.

Остается число:

7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0

Это и есть 94 = 350 - 256.

Аналогично получается результат для вычитания и умножения.

Tun double

Тип **double**, напротив, имеет более широкий диапазон значений. Приведение типов для него не требуется. Измените типы данных в программе. Не забудьте убрать слово "целых" из пояснения для пользователя.

Кроме того, дробные числа можно делить на 0, получится ∞ (бесконечность, infinity) или $-\infty$ (negative infinity). Закомментируйте проверку на равенство нулю, которую мы выполняли для целых чисел.

```
15 Console.WriteLine("Введите два числа");
16 double number1 = double.Parse(Console.ReadLine());
17 double number2 = double.Parse(Console.ReadLine());
18
19 double sum = number1 + number2;
20 Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);
21 double sub = number1 - number2;
22 Console.WriteLine("{0} - {1} = {2}", number1, number2, sub);
23 double mul = number1 * number2;
24 Console.WriteLine("{0} * {1} = {2}", number1, number2, mul);
25 double div = number1 / number2;
26 Console.WriteLine("{0} / {1} = {2}", number1, number2, div);
27 double rem = number1 % number2;
28 Console.WriteLine("{0} % {1} = {2}", number1, number2, rem);
29 /*
30 if (number2 == 0)
31 {
32     Console.WriteLine("на 0 делить нельзя");
33 }
34 else
35 {
36 }
37 */
```

В тесты следует добавить дробные значения, в том числе очень маленькие и очень большие (double поддерживает числа в диапазоне от $5,0 \cdot 10^{-324}$ до $1,7 \cdot 10^{308}$). Чтобы не писать много нулей после запятой, используем экспоненциальный формат записи:

$$1e-15 = 1 \cdot 10^{-15} = 0,000\ 000\ 000\ 000\ 001$$

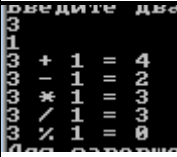
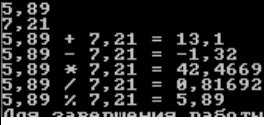
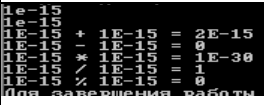
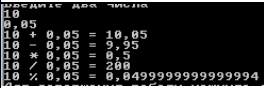
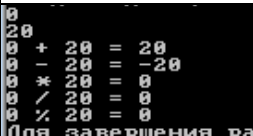
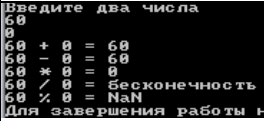
$$1,5e+10 = 1 \cdot 10^{10} = 15\ 000\ 000\ 000$$

Обратите внимание, в тексте программы дробные значения всегда пишутся через точку, а в консоли - в соответствии с системными настройками, в русскоязычной Windows - через запятую.

Результаты тестирования приведены в табл. 1.5.

Таблица 1.5

Тестирование приложения ConsoleArithmetic на типе double

№ теста	Входные данные	Ожидаемый результат	Результат программы	Примечание
1	2	3	4	5
1	3 1	$3 + 1 = 4$ $3 - 1 = 2$ $3 \cdot 1 = 3$ $3 / 1 = 3$ $3 \% 1 = 0$		Первое число больше второго, числа целые
2	5,89 7,21	$5,89 + 7,21 = 13,1$ $5,89 - 7,21 = -1,32$ $5,89 \cdot 7,21 = 42,4669$ $5,89 / 7,21 = 0,81692$ $5,89 \% 7,21 = 5,89$		Второе число больше первого, числа дробные
3	1,00E-15 15 1,00E-15	$1,00E-15 + 1,00E-15 = 1,00E-15$ $1,00E-15 - 1,00E-15 = 0$ $1,00E-15 \cdot 1,00E-15 = 1E-30$ $1,00E-15 / 1,00E-15 = 1$ $1,00E-15 \% 1,00E-15 = 0$		Числа равны и очень маленькие
4	10 0,05	$10 + 0,05 = 10,05$ $10 - 0,05 = 9,95$ $10 \cdot 0,05 = 0,5$ $10 / 0,05 = 200$ $10 \% 0,05 = 0,05$		Второе число меньше 1
5	0 20	$0 + 20 = 20$ $0 - 20 = -20$ $0 \cdot 20 = 0$ $0 / 20 = 0$ $0 \% 20 = 0$		Первое 0
6	60 0	$60 + 0 = 60$ $60 - 0 = 60$ $60 \cdot 0 = 0$ $0 / 20 = +\infty$ $0 \% 20 = NaN$		Первое положительное, второе 0

1	2	3	4	5
7	-100 0	-100 + 0 = -100 -100 - 0 = -100 -100 · 0 = 0 0 / 20 = -∞ 0 % 20 = NaN	Введите два числа -60 0 -60 + 0 = -60 -60 - 0 = -60 -60 * 0 = 0 -60 / 0 = -Бесконечность -60 % 0 = NaN Для завершения работы нажмите	Первое отрицательное, второе 0
8	-37 -12	-37 + -12 = -49 -37 - -12 = -25 -37 · -12 = 444 -37 / -12 = 3 -37 % -12 = -1	Введите два числа -100 0 -100 + 0 = -100 -100 - 0 = -100 -100 * 0 = 0 -100 / 0 = -Бесконечность -100 % 0 = NaN Для завершения работы нажмите	Отрицательные
9	9876543210 123456789	9876543210 + 123456789 = 9999999999 9876543210 - 123456789 = 9753086421 9876543210 · 123456789 = 1219326311126350000 9876543210 / 123456789 = 80,000000729000006633900060368491 9876543210 % 123456789 = 90	Введите два числа 9876543210 123456789 9876543210 + 123456789 = 9999999999 9876543210 - 123456789 = 9753086421 9876543210 * 123456789 = 1,21932631112635E+18 9876543210 / 123456789 = 80,000000729 9876543210 % 123456789 = 90 Для завершения работы нажмите любую клавишу	Большие значения
10	9,00E+205 1,00E-105	9E+205 + 1E-105 = 9E+205 9E+205 - 1E-105 = 9E+205 9E+205 · 1E-105 = 9E+100 9E+205 / 1E-105 = 9E+310 9E+205 % 1E-105 = NaN	Введите два числа 9e+205 1e-105 9e+205 + 1e-105 = 9E+205 9e+205 - 1e-105 = 9E+205 9e+205 * 1e-105 = 9E+100 9e+205 / 1e-105 = Бесконечность 9e+205 % 1e-105 = 5,59561747397225E-106 Для завершения работы нажмите любую клавишу	Очень большое и очень маленькое значения

Выводы:

1) для дробных чисел допускается вычислять остаток от деления, но результат не всегда очевиден (лучше так не делать);

2) при делении на 0, а также, если результат выходит за границы допустимого диапазона, получается либо плюс бесконечность (`double.PositiveInfinity`), либо минус бесконечность (`double.NegativeInfinity`);

3) особое значение NaN (Not a Number) означает, что результат вычислить невозможно (тесты № 6, 7, 10);

4) при очень больших и очень маленьких значениях возникают ошибки округления (тесты № 4, 9, 10).

Таким образом, если планируется работать с очень большими числами, даже если они целые, имеет смысл использовать тип `double`. Но нужно следить за возможными ошибками округления.

Tun decimal

Как видно из тестов, тип `double` может приводить к проблемам с округлением (например, в тесте № 4 вместо 0,05 получили 0,0499999999994). В некоторых ситуациях эти погрешности играют большую роль. Например, бухгалтерский баланс должен сходиться до копейки. При этом такие гигантские значения, как $1e+300$, в нем вряд ли появятся.

Поэтому для финансовых и других подобных расчетов вводится специальный тип - `decimal`, который обеспечивает точность расчетов с десятичными дробями. Он занимает много места в памяти и относительно медленно вычисляется, поэтому применять его надо строго по назначению.

Тип `decimal` не содержит значений ∞ и NaN, поэтому для него придется вернуть проверку на деление на 0. Кроме того, он не поддерживает экспоненциальную запись чисел.

```
15 Console.WriteLine( "убедитесь два числа ),  
16 decimal number1 = decimal.Parse(Console.ReadLine());  
17 decimal number2 = decimal.Parse(Console.ReadLine());  
18  
19  
20 decimal sum = number1 + number2;  
21 Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);  
22 decimal sub = number1 - number2;  
23 Console.WriteLine("{0} - {1} = {2}", number1, number2, sub);  
24 decimal mul = number1 * number2;  
25 Console.WriteLine("{0} * {1} = {2}", number1, number2, mul);  
26 if (number2 == 0)  
27 {  
28     Console.WriteLine("на 0 делить нельзя");  
29 }  
30 else  
31 {  
32     decimal div = number1 / number2;  
33     Console.WriteLine("{0} / {1} = {2}", number1, number2, div);  
34     decimal rem = number1 % number2;  
35     Console.WriteLine("{0} % {1} = {2}", number1, number2, rem);  
36 }
```

Протестируем его работу на различных значениях (табл. 1.6).

Таблица 1.6

Тестирование приложения ConsoleArithmetic на типе decimal

№ теста	Входные данные	Ожидаемый результат	Результат программы	Примечание
1	2	3	4	5
1	3 1	$3 + 1 = 4$ $3 - 1 = 2$ $3 \cdot 1 = 3$ $3 / 1 = 3$ $3 \% 1 = 0$	<pre> Введите два числа 3 1 3 + 1 = 4 3 - 1 = 2 3 * 1 = 3 3 / 1 = 3 3 % 1 = 0 </pre>	Первое число больше второго, числа целые
2	5,89 7,21	$5,89 + 7,21 = 13,1$ $5,89 - 7,21 = -1,32$ $5,89 \cdot 7,21 = 42,4669$ $5,89 / 7,21 = 0,81692$ $5,89 \% 7,21 = 5,89$	<pre> Введите два числа 5,89 7,21 5,89 + 7,21 = 13,10 5,89 - 7,21 = -1,32 5,89 * 7,21 = 42,4669 5,89 / 7,21 = 0,81692094313453536754507628 5,89 % 7,21 = 5,89 </pre>	Второе число больше первого, числа дробные
3	0,00000000 0001 0,00000000 0001	$0,000000000001 + 0,000000000001 = 0,000000000002$ $0,000000000001 - 0,000000000001 = 0$ $0,000000000001 \cdot 0,000000000001 = 0,0000000000000000000000000001$ $0,000000000001 / 0,000000000001 = 1$ $0,000000000001 \% 0,000000000001 = 0$	<pre> Введите два числа 0,0000000001 0,0000000001 0,0000000001 + 0,0000000001 = 0,0000000002 0,0000000001 - 0,0000000001 = 0,0000000000 0,0000000001 * 0,0000000001 = 0,0000000000000000000000000001 0,0000000001 / 0,0000000001 = 1 0,0000000001 % 0,0000000001 = 0,0000000000 </pre>	Числа равны и очень маленькие
4	10 0,05	$10 + 0,05 = 10,05$ $10 - 0,05 = 9,95$ $10 \cdot 0,05 = 0,5$ $10 / 0,05 = 200$ $10 \% 0,05 = 0,05$	<pre> Введите два числа 10 0,05 10 + 0,05 = 10,05 10 - 0,05 = 9,95 10 * 0,05 = 0,50 10 / 0,05 = 200 10 % 0,05 = 0,00 </pre>	Второе число меньше 1
5	60 0	$60 + 0 = 60$ $60 - 0 = 60$ $60 \cdot 0 = 0$ На 0 делить нельзя	<pre> Введите два числа 60 0 60 + 0 = 60 60 - 0 = 60 60 * 0 = 0 на 0 делить нельзя </pre>	Первое положительное, второе 0

1	2	3	4	5
6	15,99 -125	15,99 + -125 = -109,01 15,99 - -125 = 140,99 15,99 · -125 = -1998,75 15,99 / -125 = -0,12792 15,99 % -125 = -109,01	<p>Введите два числа</p> <p>15,99 -125</p> <p>15,99 + -125 = -109,01 15,99 - -125 = 140,99 15,99 * -125 = -1998,75 15,99 / -125 = -0,12792 15,99 % -125 = 15,99</p>	Отрицательные и дробные
7	9876543210 123456789	9876543210 + 123456789 = 9999999999 9876543210 - 123456789 = 9753086421 9876543210 × 123456789 = 121932631112635 0000 9876543210 / 123456789 = 80,000000729000 006633900060368 491 9876543210 % 123456789 = 90	<p>Введите два числа</p> <p>9876543210 123456789</p> <p>9876543210 + 123456789 = 9999999999 9876543210 - 123456789 = 9753086421 9876543210 * 123456789 = 1219326311126352690 9876543210 / 123456789 = 80,0000007290000066339 9876543210 % 123456789 = 90</p>	Большие зна- чения
8	1000000000 0,00000001	1000000000 + 0,00000001= 1000000000,0000 0001 1000000000,000 00001= 999999999,99999 999 1000000000 × 0,00000001= 10 1000000000 / 0,00000001= 100000000000000 000 1000000000 % 0,00000001= 0	<p>Введите два числа</p> <p>1000000000 0,00000001</p> <p>1000000000 + 0,00000001 = 1000000000,00000001 1000000000 - 0,00000001 = 999999999,99999999 1000000000 * 0,00000001 = 10,00000000 1000000000 / 0,00000001 = 100000000000000000 1000000000 % 0,00000001 = 0,00000000</p>	Очень большое и очень ма- ленькое значе- ния

Пример 3. Максимальное из трех чисел

Задание

Пользователь вводит три целых числа. Вывести значение максимального из них.

Указания к выполнению

Логические операторы

Необходимо понимать, что любое условие является логическим выражением, которое может принимать одно из двух значений `true` или `false`.

Логическое выражение можно использовать в условиях, а также сохранять в переменных типа `bool`.

Всего в программировании используются три логических оператора:

- логическое **И** (оператор `&&`): должны выполняться все условия;
- логическое **ИЛИ** (оператор `||`): должно выполняться хотя бы одно условие;
- логическое **НЕ** (оператор `!`): условие должно не выполняться.

Иначе говоря, если в цепочке логических выражений, соединенных через **И**, хотя бы одно выражение равно `false`, то все выражение в целом равно `false` и остальные условия можно уже не проверять. В логическом **ИЛИ**, наоборот, если хотя бы одно выражение в цепочке равно `true`, то все выражение равно `true` и остальные значения не имеют значения. Этот факт учитывается в вычислениях, поэтому порядок соединяемых условий может иметь значение.

Логические операторы имеют более низкий приоритет, чем арифметические операции и операторы сравнения, поэтому отдельные условия необязательно брать в скобки.

Максимальное из двух чисел

Сначала напишем более простой вариант программы - для выбора максимального из двух чисел.

Алгоритм работы программы:

1. Пользователь вводит два числа A и B .
2. Если $A \geq B$, то максимальное A , иначе максимальное B .
3. Вывести результат на экран.

4. Завершить работу программы.

Примечание. Если числа равны, то максимальным будет любое из них. Однако, если бы в задании спрашивалось, какое именно из чисел является максимальным (ответ А или В, а не просто значение), то равенство было бы отдельным вариантом (оба числа максимальные).

Создайте новый проект - консольное приложение и сохраните его под именем "MaxNumber".

Добавьте блок ввода значений переменных.

```
//ввод исходных значений
int A, B;
int.TryParse(Console.ReadLine(), out A);
int.TryParse(Console.ReadLine(), out B);
```

В этом примере мы воспользуемся другим методом для преобразования чисел в текст - TryParse. Отличие Parse и TryParse заключается в обработке неверных данных.

Если использовать Parse и ввести число в неверном формате (текст вместо числа, дробное число вместо целого, недопустимые символы, пробелы и т.п.), то программа просто вылетит с ошибкой после ввода неверного значения.

Метод TryParse при возникновении ошибки запишет в числовую переменную 0 и вернет false, но программа не вылетит. При этом TryParse не возвращает число, а получает его в качестве выходного (out) аргумента.

Сравним значения А и В в условии оператора **if**:

```
// если больше А
if (A >= B)
{
}
// иначе - больше В
else
{
}
```

Добавим переменную Max для сохранения максимального значения и присвоим ей значение в каждом блоке.

```

//максимальное значение
int Max;
// если больше A
if (A >= B)
{
    Max = A;
}
// иначе - больше B
else
{
    Max = B;
};

```

Добавим вывод результата:

```

// вывод результата
Console.WriteLine("Максимальное = " + Max);
// завершение программы
Console.ReadKey();

```

Сохраните, запустите и протестируйте программу. Используйте тестовые данные из табл. 1.7.

Неверные значения считаются равными нулю, поэтому в тесте № 6 получилось максимальное значение 1, а не 5.5, а в тесте № 7 получился 0, так как число $A < 0$.

Таблица 1.7

Тестовые примеры для выбора максимального из двух чисел

№ теста	Входные данные		Выходные данные
	A	B	Max
1	121	12	121
2	14	50	50
3	20	20	20
4	-55	-2	-2
5	aab	10	10
6	1	5.5	1
7	-1	b	0

Добавим проверку на корректность введенных значений. Если `TryParse` возвращает `true`, значит, перевод был выполнен корректно, а если `false` - то некорректно.

```

if (int.TryParse(Console.ReadLine(), out A) == false)
{
    Console.WriteLine("Неверное значение! Нужно ввести целое число.");
};

```

? Можно ли для проверки на корректность ввода сравнивать значение переменной A с нулем (A == 0)?

Однако сравнивать с true и false не принято. Это не ошибка, но просто лишнее действие: сам результат проверки уже является логическим значением. Правильнее будет использовать логическое НЕ (символ !):

```
if (!int.TryParse(Console.ReadLine(), out A))
{
    Console.WriteLine("Неверное значение! Нужно ввести целое число.");
};
```

Если вы запустите программу, то увидите, что сообщение будет выведено, но программа продолжит работу. То есть нам необходимо выполнить все дальнейшие действия, если TryParse возвращает True, а в противном случае только вывести сообщение и завершить программу.

Для экстренного завершения работы можно использовать команду return, но тогда программа сразу закроется, и мы не успеем увидеть сообщение об ошибке. Кроме того, такой код считается "грязным", так как часто затрудняет чтение и анализ чужих программ.

```
if (!int.TryParse(Console.ReadLine(), out A))
{
    Console.WriteLine("Неверное значение! Нужно ввести целое число.");
    return;
};
```

Поэтому мы перенесем сообщение об ошибке в блок else, а остальную программу - внутрь блока if. В конце останется только нажатие клавиши.

```
if (int.TryParse(Console.ReadLine(), out A))
{
    int.TryParse(Console.ReadLine(), out B);
    //максимальное значение
    int Max;
    // если больше A
    if (A >= B)
    {
        Max = A;
    }
    // иначе - больше B
    else
    {
        Max = B;
    }
    // вывод результата
    Console.WriteLine("Максимальное = " + Max);
}
else
{
    Console.WriteLine("Неверное значение! Нужно ввести целое число.");
};
// завершение программы
Console.ReadKey();
```

Добавим аналогичную проверку для B.

```

if (int.TryParse(Console.ReadLine(), out A))
{
    if (int.TryParse(Console.ReadLine(), out B))
    {
        //максимальное значение
        int Max;
        // если больше A
        if (A >= B)
        {
            Max = A;
        }
        // иначе - больше B
        else
        {
            Max = B;
        }
        // вывод результата
        Console.WriteLine("Максимальное = " + Max);
    }
    else
    {
        Console.WriteLine("Неверное значение! Нужно ввести целое число.");
    }
}
else
{
    Console.WriteLine("Неверное значение! Нужно ввести целое число.");
}
};

```

? Как объединить проверки для A и B в один if? Будет ли при этом программа запрашивать ввод B, если A введено неверно?

Протестируйте работу программы.

Тернарный оператор ?:

Другой способ выполнения проверки - использование тернарной операции <условие> ? <значение_если_true> : <значение_если_false> (тернарная - с тремя операндами).

Эту операцию можно использовать в вычислениях наравне с обычными операциями сложения, умножения и т.д. Однако рекомендуется не злоупотреблять ей, чтобы не усложнить прочтение кода.

Тогда проверка значений A и B сократится до одной строчки:

```

//максимальное значение
int Max = (A >= B) ? A : B;
// вывод результата
Console.WriteLine("Максимальное = " + Max);

```

Читается: если A больше или равно B, то присвоить A, иначе присвоить B.

Закомментируйте, но не удаляйте вариант кода с if. Протестируйте работу оператора? : .

Третье число

Расширим программу до сравнения трех переменных. Вернемся к варианту программы с `if`: раскомментируйте его, а `?:` закомментируйте.

Добавьте ввод значения переменной `C` по аналогии с `A` и `B`.

```
int A, B, C;  
if (int.TryParse(Console.ReadLine(), out A) && int.TryParse(Console.ReadLine(), out B)  
    && int.TryParse(Console.ReadLine(), out C))
```

Добавим проверку значений. Теперь, если $A > B$, его еще нужно сравнить с `C`:

```
// если A >= B  
if (A >= B)  
{  
    // и если A >= C  
    if (A >= C)  
    {  
        // то максимальное - A  
        Max = A;  
    }  
    // иначе - C > A  
    else  
    {  
        // максимальное - C  
        Max = C;  
    }  
}
```

Аналогично для `B`:

```
// иначе - больше B  
else  
{  
    // и если B >= C  
    if (B >= C)  
    {  
        // то максимальное - B  
        Max = B;  
    }  
    // иначе - C > B  
    else  
    {  
        // максимальное - C  
        Max = C;  
    }  
};
```

Протестируйте правильность работы программы. Самостоятельно составьте таблицу с тестовыми значениями.

Другие варианты записи условий

Другой вариант реализации проверок через логическое И в условиях:

```
// если A больше остальных
if ((A >= B) && (A >= C))
{
    Max = A;
}
// иначе, если B больше остальных
else if ((B >= A) && (B >= C))
{
    Max = B;
}
// иначе - остается C
else
{
    Max = C;
};
```

Этот код значительно короче и читабельнее, но немного затратнее по времени выполнения: если максимальное значение C, то необходимо выполнить четыре разных сравнения, причем $(A \geq B)$ и $(B \geq A)$ фактически дублируют друг друга. В предыдущем варианте, какое бы число не было максимальным, всегда выполняется два сравнения.

При однократном запуске разница составит менее миллисекунды, но если этот код вызывается тысячи раз, разница будет заметной.

Наконец, третий вариант - сравнить C не напрямую с A и B, а с максимальным из них (максимальное из двух значений мы уже находили):

```
//максимальное значение
int Max;
// если больше A
if (A >= B)
{
    Max = A;
}
// иначе - больше B
else
{
    Max = B;
};
// если C больше максимального из A, B, то C - максимальное
if (C >= Max)
{
    Max = C;
};
```

В этом варианте будет выполнено два сравнения, но и значение переменной Max будет присвоено дважды. Таким образом, этот код тоже немного медленнее первого варианта.

После этого вернитесь к варианту кода, где мы остановились на сравнении двух чисел с помощью тернарной операции?: (раскомментируйте его, а вариант с `if` закомментируйте).

Добавьте ввод значения `C`, как было показано ранее.

Реализуем поиск максимума по аналогии с последним вариантом `if`, когда мы сначала находим большее из `A` и `B`, а потом сравниваем его с `C`:

```
//максимальное значение
int Max = (A >= B) ? A : B;
Max = (Max >= C) ? Max : C;
```

? Можно ли пропустить скобки в условии? А в `if`?

Этот код можно сократить до одной строки. `C#` поддерживает операции присваивания "в середине" арифметического выражения. Поэтому первый раз можно присвоить значение `Max` прямо в проверке условия:

```
//максимальное значение
int Max = ((Max = (A >= B) ? A : B) >= C) ? Max : C;
```

Как видите, даже такую простую задачу можно решить множеством способов, отличающихся по длине кода, его читабельности и скорости выполнения.

Пример 4. Единицы измерения информации

Пользователь вводит дробное число - размер файла и указывает единицу измерения (Кб, Мб, Гб, Тб, Пб). Необходимо вывести, сколько это байт.

Указания к выполнению

В данной программе воспользуемся оператором множественного выбора `switch`.

Создайте новый проект "FileSize". Напишите код для ввода двух значений: размера файла (числовой, дробный с плавающей запятой) и единицы измерения (строка). Не забудьте сразу добавить `ReadKey`.

```
13 Console.WriteLine("Перевод размера файла в байты");
14 Console.WriteLine(".....");
15 Console.WriteLine("Введите исходный размер файла:");
16 double originalSize = double.Parse(Console.ReadLine());
17 Console.WriteLine("Введите единицу измерения (Кб, Мб, Гб, Тб и Пб):");
18 string unit = Console.ReadLine();
19
20 Console.Write("Нажмите любую клавишу...");
21 Console.ReadKey();
```

Оператор множественного выбора

Оператор условного выбора `if` хорошо подходит для случаев, когда нужно проверить сложное условие или выбрать один из 2-3 вариантов. Когда вариантов много, а проверяется значение одной переменной, лучше использовать оператор множественного выбора **`switch`**. Его синтаксис:

```
switch (/*проверяемая_переменная*/)
{
    case /*значение_1*/: /*действия*/; break;
    case /*значение_2*/: /*действия*/; break;
    case /*значение_3*/: /*действия*/; break;
    //...
    default: /*действия*/; break;
};
```

Каждый вариант начинается с ключевого слова `case` и заканчивается словом `break`. В конце может присутствовать вариант по умолчанию `default`, который сработает, если ни один из предыдущих вариантов не подошел.

Нам необходимо проверить значение переменной `unit`. Запишем каркас оператора `switch`.

```
17 |         string unit = Console.ReadLine();
18 |
19 |         switch (unit)
20 |         {
21 |             case "КБ": break;
22 |             case "МБ": break;
23 |             case "ГБ": break;
24 |             case "ТБ": break;
25 |             case "ПБ": break;
26 |             default: break;
27 |         }
```

Каждая следующая единица измерения в 1024 раза больше предыдущей. Результат перевода сохраним в переменной `byteSize`. Поскольку ее значение может быть очень большим, для нее также выберем тип **`double`**. Если пользователь ввел неверную единицу измерения, будем считать `byteSize` равным `NaN`.

```
19 |         double byteSize;
20 |         switch (unit)
21 |         {
22 |             case "КБ": byteSize = originalSize * 1024; break;
23 |             case "МБ": byteSize = originalSize * 1024 * 1024; break;
24 |             case "ГБ": byteSize = originalSize * 1024 * 1024 * 1024; break;
25 |             case "ТБ": byteSize = originalSize * 1024 * 1024 * 1024 * 1024; break;
26 |             case "ПБ": byteSize = originalSize * 1024 * 1024 * 1024 * 1024 * 1024; break;
27 |             default: byteSize = double.NaN; break;
28 |         }
```

Осталось определиться с форматом вывода результата.

```
28 |     }
29 |
30 |     Console.WriteLine("{0} {1} = {2} байт", originalSize, unit, byteSize);
31 |
32 |     Console.WriteLine("Нажмите любую клавишу.");
```

На этом разработку приложения завершим.

- ? Как будет работать программа, если убрать операторы break?
- ? Как будет работать программа, если поставить ветвь default перед всеми case?

Тестирование приложения

Протестируем работу приложения для всех единиц измерения и для неверной единицы измерения (табл. 1.8). Проверим целые и дробные значения. Предполагается, что размер файла не может быть отрицательным.

Таблица 1.8

Тестовые примеры для перевода единиц измерения информации

№ п/п	Входные данные	Ожидаемый результат	Результат программы	Примечание
1	1 Кб	1 Кб = 1024 байта	<pre> Перевод размера файла в байты Введите исходный размер файла: 1 Введите единицу измерения: Кб 1 Кб = 1024 байт Нажмите любую клавишу... </pre>	Целое число, Кб
2	2,45 МБ	2,45 МБ = 2569011,2 байта	<pre> Перевод размера файла в байты Введите исходный размер файла: 2,45 Введите единицу измерения: МБ 2,45 МБ = 2569011,2 байт Нажмите любую клавишу... </pre>	Дробный размер, МБ, дробные байты
3	20,125 Гб	20,125 Гб = 21102592 байта	<pre> Введите исходный размер файла: 20,125 Введите единицу измерения: Гб 20,125 Гб = 21102592 байт Нажмите любую клавишу... </pre>	Дробный размер, Гб, байты целые
4	1024 Тб	1024 Тб = 1099511627776 байтов	<pre> Перевод размера файла в байты Введите исходный размер файла: 1024 Введите единицу измерения: Кб 1024 Тб = 1099511627776 байт Нажмите любую клавишу... </pre>	Целый размер, Тб
5	18,5 Пб	18,5 Пб = 20340965113856 байтов	<pre> Перевод размера файла в байты Введите исходный размер файла: 18,5 Введите единицу измерения: Кб 18,5 Пб = 20340965113856 байт Нажмите любую клавишу... </pre>	Дробный размер, Гб, байты целые, большое число
6	1 mb	1 mb = NaN байтов	<pre> Введите исходный размер файла: 1 Введите единицу измерения: mb 1 mb = NaN байт Нажмите любую клавишу... </pre>	Неверная единица

? Как изменить программу, чтобы для неверных единиц измерения вывелось сообщение об ошибке?

Задачи для самостоятельного решения

Задача 1.1. Дни, часы, минуты

На вход в программу подается количество минут, прошедших с какого-то события (например, с загрузки компьютера). Вывести, сколько это дней, часов, минут. Если срок меньше суток, то дни не выводить.

Подсказка: используйте / и %.

Примеры с форматом вывода представлены в табл. 1.9

Таблица 1.9

Тестовые примеры к задаче 1.1

Вход	Выход
63	01:03
1	00:01
192	03:12
3965	2 дня 18:05
1481	1 день 00:41

Задача 1.2. Скидка

Сумма покупки составляет value рублей. Если value больше 1000, то предоставляется скидка 15 %. Вывести на экран сумму покупки с учетом скидки либо сообщение о том, что скидка не предоставляется.

Задача 1.3. Кассовый чек v.0.0.1

Вводится три числа: цена товара, его количество (может быть дробным, например, 2,85 кг картофеля) и полученная оплата. Необходимо определить, нужно ли выдать сдачу или, наоборот, покупатель должен доплатить еще. Вывести стоимость товара и сумму сдачи или необходимой доплаты.

Примеры:

Введите цену и количество купленного товара.

Цена: 35

Кол-во: 1

Сумма: 35,00

Введите полученную сумму наличных.

Получено: 35,00

Сдача: 0,00

Введите цену и количество купленного товара.

```
Цена: 68,54
Кол-во: 2,256
Сумма: 154,6
```

```
Введите полученную сумму наличных.
Получено: 200
Сдача: 45,37
```

```
Введите цену и количество купленного товара.
Цена: 789,9
Кол-во: 5
Сумма: 3949,5
```

```
Введите полученную сумму наличных.
Получено: 3000
Доплата: 949,50
```

Задача 1.4. Курсы валют

В программе константами заданы коэффициенты для перевода одной валюты в другую (рубли в евро, доллары, фунты, йены). Пользователь вводит сумму в рублях и выбирает, в какую валюту хочет их перевести. Программа переводит сумму в выбранную валюту с округлением до двух знаков после запятой.

Задача 1.5. Избавиться от if

Даже если задача звучит как условие "если", это еще не значит, что использовать `if` обязательно. Иногда его можно заменить формулой в одну строку.

В программе присутствует фрагмент кода следующего вида:

```
//если a нечетное
if (a % 2 == 1)
{
    //то b сделать следующим после a четным числом
    b = a + 1;
}
else
{
    //иначе (a четное) b равно a
    b = a;
}
```

Как записать данный фрагмент без `if`, в одну формулу? Использовать тернарный оператор?: или `switch` тоже нельзя, необходимо обойтись совсем без условия.

Переменные `a` и `b` имеют тип `int`.

Задача 1.6. Свойства файла

Пользователь вводит полный путь к файлу. Проверить, существует ли такой файл. Если существует - то выведите его размер, тип, дату и время последнего изменения.

Самостоятельно познакомьтесь с классом `FileInfo` и его методами.

Контрольные вопросы

1. В каких случаях в современном программировании применяют консольные приложения?
2. Как осуществляется ввод-вывод данных в консоль?
3. Перечислите основные типы данных в языке C#.
4. Какой тип данных лучше всего подходит для хранения:
 - а) денежных величин;
 - б) количества студентов в группе;
 - в) фамилии и имени пользователя;
 - г) доли затрат в общей сумме?
5. Что такое конкатенация? Каким оператором она обозначается?
6. Перечислите арифметические операторы в C#.
7. Из каких частей состоит оператор ветвления `if`?
8. Как записываются операторы сравнения?
9. Как записываются логические операторы?
10. В каких случаях вместо оператора `if` целесообразно применять тернарный оператор `"?:"`?
11. Что такое оператор множественного выбора?
12. Можно ли заменить оператор `switch` на `if`?

2. ОСНОВЫ ЯЗЫКА C#. СТРОКИ И СИМВОЛЫ. ЦИКЛЫ. МАССИВЫ

Символьный и строковый тип

В C# два базовых типа предназначены для работы с текстом:

`char` - отдельный **символ**, пишется в одинарных кавычках (апострофах), например, 'a', '5', 'И', '+', 'π', '\t', '\n', а также специальные клавиши - Esc, Del, Backspace, F1-F12, стрелки и др;

`string` - **строка**, состоящая из последовательности символов, пишется в двойных кавычках, например, "Мария", "ул. Советской Армии, д. 141".

Строка является, по сути, массивом символов. Поэтому мы всегда можем извлечь из нее отдельный символ по его номеру. Например,

```
string city = "Сомаpa"; // строка с опечаткой
char letter = city[0]; // letter = 'C'
```

Но это массив только для чтения, присваивать значения отдельным символам нельзя.

```
city[1] = 'a'; // ОШИБКА! массив только для чтения
```

Также символ нельзя напрямую конвертировать в строку, необходимо использовать метод `ToString()`.

```
string cat1 = 'к'; // ОШИБКА! нельзя преобразовать
string cat2 = 'к'.ToString(); // так правильно
// и так можно
```

```
string cat3 = "КОТ";
cat3 = cat3[1].ToString(); // cat3 = "к"
```


Специальные символы можно задавать с помощью так называемых **escape-последовательностей**, например:

\n - новая строка

\t - табуляция

\u2192 - символ Unicode с кодом 2192.

С точки зрения программы, отдельный символ является даже не строкой, а числом (именно поэтому символы нельзя явно преобразовывать в строки). Это число - код символа в **таблице Unicode** (Юникод, UTF-8).

Узнать код символа и то, где он находится в таблице Unicode, можно через Word (на ленте "Вставка" → "Символ" →  "Другие символы..."? шрифт "(обычный текст)"). Окно с символами показано на рис. 2.1. Обратите внимание, что коды символов здесь записываются в шестнадцатеричном формате.

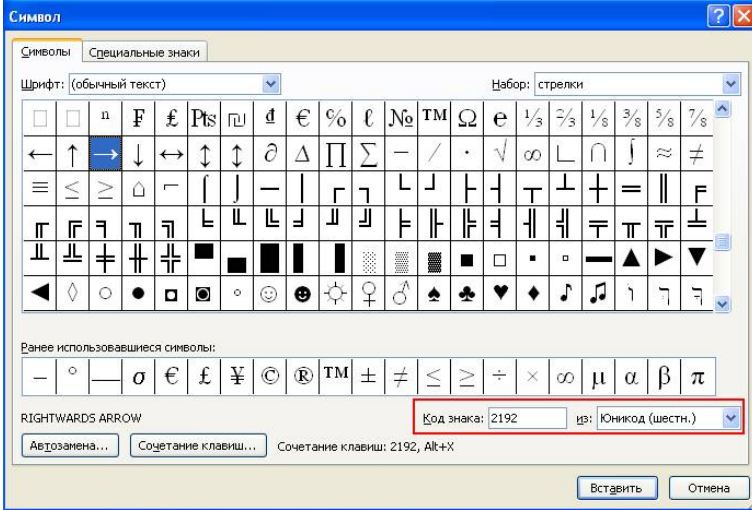


Рис. 2.1. Таблица символов Unicode с кодами в шестнадцатеричной системе

Поэтому символы, как и числа, можно сравнивать между собой. Чем больше код символа, тем больше символ. А поскольку в Unicode все символы записываются по алфавиту, то чем дальше символ в алфавите, тем он "больше". Заглавные буквы "меньше" строчных. Из всех алфавитов первой идет латиница, поэтому ее буквы "меньше" всех остальных. В других случаях результат сравнения не так очевиден.

Например:

- 'a' < 'b' < 'c' < ... < 'z'
- 'я' > 'ю' > ... > 'в' > 'б' > 'а'
- '0' < '1' < ... < '9'
- 'b' > '1'
- 'A' < 'a'
- 'я' > 'z'
- 'ч' > 'л'
- '!' < '0'
- '=' < '≠'
- 'Δ' > '©'

Методы строк

Конкатенацию (сцепку, склеивание) строк можно выполнить двумя способами:

- оператором +;
- методом `string.Concat(str1, str2, ...)`.

Пример:

```
string word1 = "прекрасный";
string word2 = "осенний";
string word3 = "день";
string phrase1 = word1 + " " + word2 + " " + word3;
// phrase1 = "прекрасный осенний день"
string phrase2 = string.Concat(word2, " ", word3, "
", word1);
// phrase2 = "осенний день прекрасный"
```

Можно выполнять конкатенацию символов, в результате получится строка:

```
string imya = "Аполлон";
string otchestvo = "Митрофанович";
string familiya = "Сатанеев";
string fio = imya[0] + otchestvo[0] + familiya[0];
// 'А' + 'М' + 'С' = "АМС"
```

Для строк существует множество полезных встроенных методов.

`ToUpper()` - перевод строки в верхний регистр (все буквы заглавные).

`ToLower()` - перевод строки в нижний регистр (все буквы строчные).

```
string sseu = "Стэу";
Console.WriteLine(sseu.ToUpper()); // "СТЭУ"
Console.WriteLine(sseu.ToLower()); // "стэу"
```

`Trim()` - обрезка пробелов в начале и конце строки;

```
string adress = " ул. Победы, 58-12 \n";
adress = adress.Trim(); // adress = "ул. Победы,
58-12"
```

? Выясните, какие еще символы, кроме пробелов, обрезает `Trim`.

Длину строки можно узнать так же, как и длину массива - через свойство `Length`.

```
string s = "абвгде";
int len = s.Length; // len = 6
```

`Substring(с_какого_символа, сколько_символов)` - выделение подстроки, начиная с указанного символа (иначе говоря, об-

резка строки). Второй аргумент (сколько символов) можно пропустить, тогда строка обрежется до конца.

```
string hW = "Hello, world!";
string h = hW.Substring(0, 5); // h = "Hello"
string w = hW.Substring(7, 5); // w = "world"
string w2 = hW.Substring(7); // w2 = "world!"
```

Это наиболее часто используемые, но не единственные методы строк.

Цикл for

Лучше всего для перебора последовательных значений подходит цикл for. Его называют **цикл-счетчик** или **цикл с параметром**. Он перебирает значения некоторой переменной от начального до конечного с заданным шагом.

```
for (счетчик = начальное значение; конечное
значение; шаг)
{
    //тело_цикла
}
```

Один повтор цикла называется **итерацией**.

В начале итерации проверяется, не достиг ли счетчик конечного значения (можно записать любое условие). В конце итерации значение счетчика увеличивается или уменьшается на заданный шаг (можно использовать любую формулу).

Например, выведем в консоль значения от 1 до 10:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine(i);
}
```

Рассмотрим по частям:

int i = 1 - переменная-счетчик. Она объявлена и существует только внутри данного цикла for. Начальное значение равно 1.

i <= 10 - конечное значение, или условие останова цикла. **for** будет повторяться, только пока **i** меньше или равно 10. Как только **i** станет равно 11, цикл прекратится. Условие проверяется каждый раз в начале цикла.

i++ - шаг цикла. Такая запись называется **"инкремент"** и означает увеличение **i** на 1. Можно было бы записать **i = i + 1**, но инкремент короче.

Составное присваивание

В синтаксисе С-подобных языков присутствует присваивание, которое изменяет существующее значение переменной, а не записывает в нее новое. Операторы составного присваивания существуют для каждой арифметической операции, а также для действий инкремента и декремента (табл. 2.1).

Таблица 2.1

Операторы составного присваивания

Составное присваивание	Пример	Аналог обычного присваивания	Пояснение
++	x++	x = x + 1	Инкремент , увеличить x на 1
--	x--	x = x - 1	Декремент , уменьшить x на 1
+=	x += 2	x = x + 2	Увеличить x на 2
-=	x -= 3	x = x - 3	Уменьшить x на 3
*=	x *= 5	x = x * 5	Увеличить x в 5 раз
/=	x /= 4	x = x / 4	Уменьшить x в 4 раза
%=	x %= 10	x = x % 10	Ограничить x значениями от 0 до 10

Составное присваивание - всего лишь более короткая запись обычного присваивания. Предпочтительно использовать составное присваивание, если это возможно.

Конкатенацию тоже можно записать через составное присваивание, при этом добавленная строка запишется в конец существующей:

```
string address = "г. Самара, ";
address += "ул. Советской Армии, ";
address += "141";
// address = "г. Самара, ул. Советской Армии, 141"
```

Циклы с предусловием и постусловием

Часто в цикле требуется не просто перебрать значения, а работать, пока выполняется какое-то условие (порой довольно сложное).

Тогда используются циклы с предусловием и постусловием.

Цикл с предусловием (while):

```
while (условие)
{
    //тело_цикла
}
```

Цикл с постусловием (do-while):

```
do
{
    //тело_цикла
```

```
} while (условие);
```

Оба цикла выполняются, пока условие остается истинным. Разница лишь в том, что в цикле с предусловием оно проверяется в начале итерации, а с постусловием - в конце.

Поэтому цикл `while` может не выполниться ни разу, а цикл `do-while` всегда выполняется как минимум один раз. Цикл `do-while` применяют, когда в условии используется какое-то значение, которое вычисляется только в теле цикла.

Цикл `for` всегда можно заменить на цикл `while`. Например, вывод в консоль значений от 0 до n:

```
for (int i = 0; i <= n; i++)
{
    Console.WriteLine(i);
}
```

В виде цикла `while`:

```
int i = 0;
while (i <= n)
{
    Console.WriteLine(i);
    i++;
}
```

Аналогичный `do-while`:

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while (i <= n);
```

Или можно инкремент выполнить прямо в условии

```
int i = 0;
do
{
    Console.WriteLine(i);
} while (i++ <= n);
```

На практике, если можно применить цикл `for`, лучше его и применять.

Математические расчеты

В данной теме нам потребуется математика немного глубже, чем арифметические операции.

Все встроенные математические функции и константы находятся в системном классе `Math`:

- модуль (абсолютная величина - без знака): `Math.Abs`;
- знак числа: `Math.Sign`;
- квадратный корень: `Math.Sqrt`;
- округление: `Math.Ceiling`, `Math.Floor`, `Math.Round`, `Math.Truncate`;
- возведение в степень: `Math.Pow`;
- тригонометрические функции: `Math.Sin`, `Math.Cos`, `Math.Tan`;
- обратные тригонометрические функции: `Math.ASin`, `Math.ACos`, `Math.Atan`, `Math.ATan2`;
- экспонента и логарифмы: `Math.Exp`, `Math.Log`, `Math.Log10`;
- константы `Math.PI`, `Math.E`.

Подробнее с синтаксисом методов можно познакомиться в интеллектуальной подсказке IntelliSense.

Пример 1. Фамилия, имя, отчество

Пользователь вводит три строки: имя, отчество и фамилию. Программа должна вывести следующие комбинации в указанном регистре:

- 1) Фамилия, Имя Отчество;
- 2) Имя Отчество ФАМИЛИЯ;
- 3) И. О. Фамилия;
- 4) фам_им_от

Отчество может отсутствовать: если в качестве отчества указана пустая строка, то соответствующие части следует пропустить.

Примеры:

Имя: Олег	Имя: джон	Имя: аНтОн
Отчество: Алексеевич	Отчество:	Отчество: семеныЧ
Фамилия: Кузнецов	Фамилия: смит	Фамилия: Шпак
Результат:	Результат:	Результат:
Кузнецов, Олег	Смит, Джон	Шпак, Антон
Алексеевич	Джон СМИТ	Семеныч
Олег Алексеевич	Д. Смит	Антон Семеныч
КУЗНЕЦОВ	сми_дж	ШПАК
О. А. Кузнецов		А. С. Шпак
куз_ол_ал		шпа_ан_се

Указания к выполнению

Создайте консольное приложение с именем "ConsoleFIO".
Добавьте переменные для ввода имени, отчества и фамилии:

```
12     {
13         Console.Write("Введите имя: ");
14         string imya = Console.ReadLine();
15         Console.Write("Введите отчество: ");
16         string otchestvo = Console.ReadLine();
17         Console.Write("Введите фамилию: ");
18         string familiya = Console.ReadLine();
19
20         Console.ReadKey();
21     }
```

Конкатенация

Сформируем первую комбинацию "Фамилия, Имя Отчество". Выглядит довольно просто, нужно всего лишь выполнить конкатенацию строк, вставив между ними запятую и пробелы.

```
20         // Фамилия, Имя Отчество
21         Console.WriteLine(familiya + ", " + imya + " " + otchestvo);
```

Верхний регистр

Во второй комбинации меняем последовательность, кроме того, фамилию нужно вывести в верхнем регистре (заглавными буквами).

```
23         // Имя Отчество ФАМИЛИЯ
24         Console.WriteLine(imya + " " + otchestvo + " " + familiya.ToUpper());
```

Отдельный символ строки

В третьем пункте из имени и отчества берутся только первые символы. Так как строка - это массив символов, берем их по индексу [0]. Не забудьте добавить точки после инициалов.

```
26         // И. О. Фамилия
27         Console.WriteLine(imya[0] + ". " + otchestvo[0] + ". " + familiya);
```

Подстрока

Последняя комбинация - генератор логина из ФИО - наиболее сложная. Здесь требуется взять первые три буквы фамилии, две буквы имени и две буквы отчества. Здесь лучше воспользоваться методом Substring.

```
29         // фам_им_от
30         Console.WriteLine(familiya.Substring(0, 3) + "_"
31                             + imya.Substring(0, 2) + "_"
32                             + otchestvo.Substring(0, 2));
--
```


Нижний регистр

Кроме того, в логине все буквы должны быть в нижнем регистре. Но применять метод `ToLower()` к каждой части отдельно - это слишком длинно. Лучше возьмем всю получившуюся строку в скобки и применим метод `ToLower()` к ней целиком.

```
29 // фам_им_от
30 Console.WriteLine((familiya.Substring(0, 3) + " "
31 + imya.Substring(0, 2) + " ")
32 + otchestvo.Substring(0, 2)).ToLower());
```

Протестируйте работу программы (рис. 2.2).

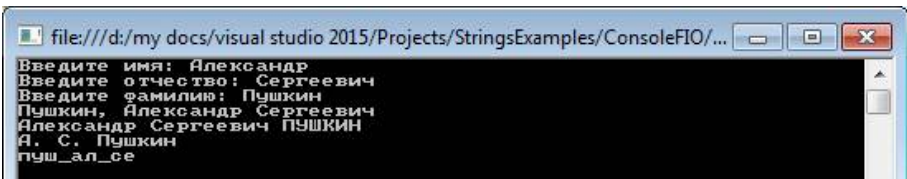


Рис. 2.2. Тестирование программы ConsoleFIO

Исправить регистр

Но нужно учесть, что не все пользователи следуют правилам русского языка. Пользователь может написать имя строчными (маленькими) буквами или даже перемешать строчные и заглавные буквы. А вывести результат требуется в формате "первая буква заглавная, остальные строчные".

Лучше всего сразу исправить ФИО в переменных, чем каждый раз менять регистр внутри `WriteLine`.

Для преобразования регистра нам потребуется "разрезать" имя на первую букву (заглавная) и все остальные буквы (строчные).

Начнем с первой заглавной буквы.

`C#` позволяет использовать "многоярусные" конструкции, т.е. нам не нужно писать в две строки:

```
string i0 = imya[0].ToString();
i0 = i0.ToUpper();
```

а можно сразу:

```
imya[0].ToString().ToUpper()
```

Чтобы вырезать из имени все буквы, кроме первой (строго говоря, нулевой), воспользуемся `Substring`:

```
imya.Substring(1)
```

Таким образом, получим конструкцию:

```

13 | Console.Write("Введите имя: ");
14 | string imya = Console.ReadLine();
15 | imya = imya[0].ToString().ToUpper() + imya.Substring(1).ToLower();

```

По аналогии самостоятельно исправьте регистр отчества и фамилии. Протестируйте работу программы (рис. 2.3).

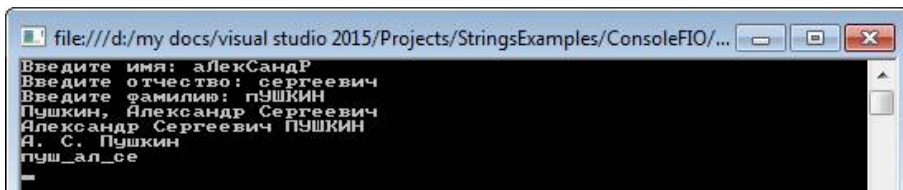


Рис. 2.3. Тестирование исправления регистра символов в ConsoleFIO

? Иногда встречаются двойные имена и фамилии, например, Джеймс-Оливер, Мария-Антуанетта, Салтыков-Щедрин, Петров-Водкин. Подумайте, как сделать заглавной первую букву каждой части.

Проверка условия

Самостоятельно добавьте проверку, было ли введено отчество (оно не должно быть пустой строкой). Если нет, то пропустите соответствующие части строк, как показано в примерах (рис. 2.4).

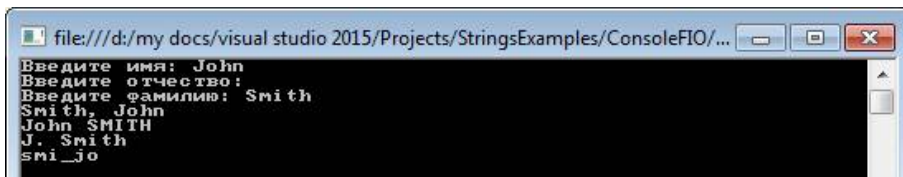


Рис. 2.4. Результат работы ConsoleFIO при отсутствии отчества

Пример 2. Простой перебор значений

Пользователь вводит целое положительное число n . Вывести:

- числа от 0 до $(n - 1)$;
- числа от $-n$ до n включительно;
- значения дробей $1/n; 1/(n-1); 1/(n-2) \dots 1/2; 1/1$;
- 10 первых чисел, которые делятся на n нацело, начиная с самого n (числа, кратные n);
- степени 2, не превышающие n , начиная с $2^0 = 1$;

– числовую лесенку по аналогии с примером, но если $n > 9$, то ограничиться ступенькой от 9 до 0.

Пример:

```
Введите целое положительное число:
n = 5

числа от 0 до (n-1):
0 1 2 3 4
числа от -n до n:
-5 -4 -3 -2 -1 0 1 2 3 4 5
дроби от 1/n до 1/1:
1/5 = 0,2
1/4 = 0,25
1/3 = 0,333333333333333
1/2 = 0,5
1/1 = 1
Первые 10 чисел, кратных n:
5 10 15 20 25 30 35 40 45 50
Степени 2, не превышающие n:
1 2 4
Числовая лесенка:
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
```

Указания к выполнению

Проекты и решения

В прошлой теме мы разобрали структуру кода программы. Но реальные приложения редко состоят из одного файла, а пакеты программ могут содержать множество приложений плюс библиотеки, базы данных, веб-страницы и т.д.

В VisualStudio **проектом** называется совокупность всех файлов одного приложения (исходные коды, файлы рисунков, подключения к внешним источникам данных и др.). Именно проект компилируется, когда вы нажимаете на кнопку "Пуск".

Решение - это несколько проектов, которые относятся к одной задаче и разрабатываются одновременно. Решение позволяет быстро переключаться между проектами, не закрывая их.

В данной теме все примеры реализуем как отдельные приложения, а всю тему объединим в одно решение.

Откройте предыдущий пример, если вы его уже закрыли. В окне обозревателя решений переименуйте *решение* "ConsoleFIO" в "StringsAndArrays" (строки и массивы). Имя *проекта* изменять не нужно (рис. 2.5).

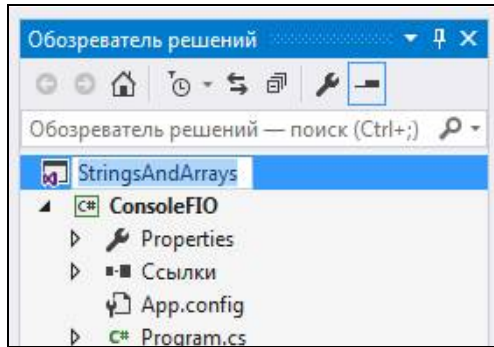


Рис. 2.5. Изменение имени решения

Теперь создадим новый проект "SimpleSequences" (простые последовательности) и при создании добавим его в существующее решение (рис. 2.6).

? Посмотрите, где будет создана папка проекта.

Теперь в обозревателе решений содержится 2 проекта. Но активен (т.е. будет запускаться) по-прежнему проект "ConsoleFIO" (его имя выделено жирным в списке).

Переключитесь на проект "SimpleSequences" (правый клик по его имени - "⚙️ Назначить автозагружаемым проектом"). Также активный проект можно выбрать в выпадающем списке возле кнопки запуска решения.

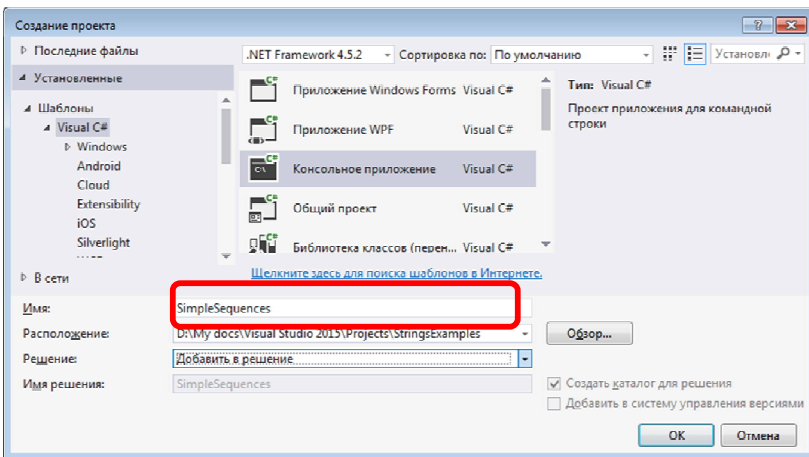


Рис. 2.6. Добавление проекта в ранее созданное решение

Обратите внимание, в обоих проектах файл с исходным кодом программы по умолчанию называется "Program.cs". Чтобы не путаться, переименуйте эти два файла в "ProgramFIO.cs" и "ProgrammSeq.cs", соответственно (рис. 2.7). Аналогично переименуйте пространства имен в файлах

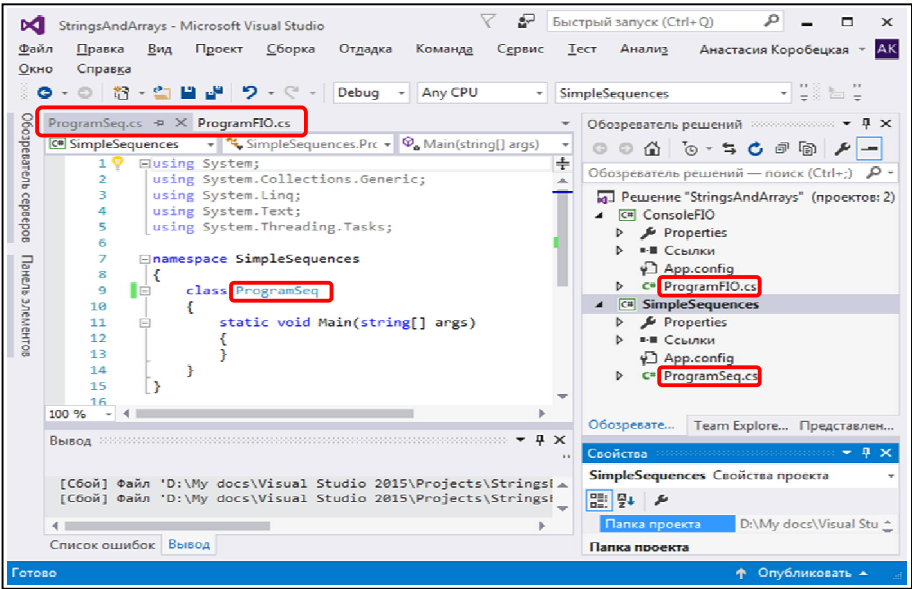


Рис. 2.7. Переименование файлов проектов в одном решении

Добавьте код для ввода целого числа n. Используйте Write вместо WriteLine, чтобы вывести "n = " в той же строке, где пользователь введет значение.

```
11     static void Main(string[] args)
12     {
13         Console.WriteLine("Введите целое положительное число");
14         Console.Write("n = ");
15         int n = int.Parse(Console.ReadLine());
```

Точка останова

Использовать ReadKey мы в этот раз не будем - обычно консольные программы не требуют нажатия клавиши для завершения работы.

Вместо этого кликните по серому полю в строке с закрывающей фигурной скобкой. Там появится красный кружок - это **точка останова (breakpoint)**, как показано на рис. 2.8.

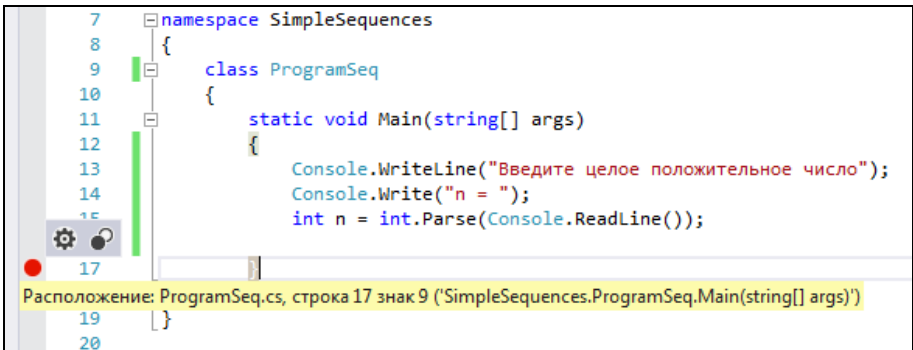


Рис. 2.8. Создание точки останова

Примечание. Не путайте точку останова программы с условием останова цикла.

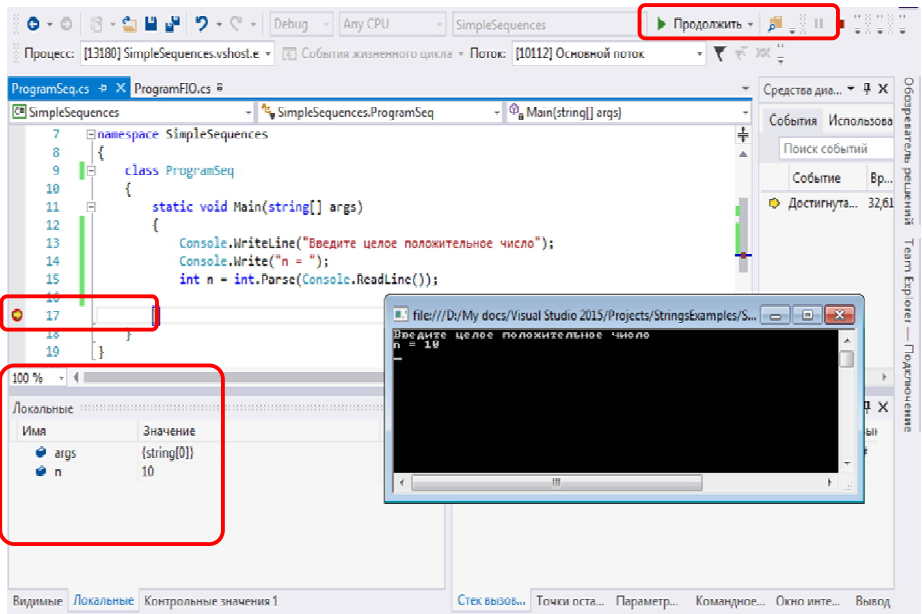


Рис. 2.9. Просмотр значений переменных при прерывании выполнения программы

Точки останова используются для отладки программы, их можно поставить в любой части кода. Как только программа дойдет до точки оста-

нова, ее выполнение будет поставлено на паузу и управление будет передано Visual Studio. Точки останова очень помогают при поиске ошибок.

Запустите приложение (убедитесь, что активен правильный проект!). После ввода значения n программа прервется и переключится на Visual Studio, но окно консоли будет доступно для просмотра.

Строка, на которой остановилась программа, отмечается желтым маркером (эта строка еще не выполнена).

Можно продолжить выполнение программы (F5), прервать его (Shift+F5) или продолжить по одной строке (F11), также см. меню "Отладка".

Можно просмотреть значения всех переменных (и убедиться в их правильности или найти ошибки), как показано на рис. 2.9.

Сейчас нам нужно просто завершить программу, поэтому нажимаем F5 или Shift+F5.

Примечание. Не забывайте завершить работу консоли - редактирование кода программы в режиме отладки запрещено.

Числа от 0 до (n-1)

Запишем первый цикл для вывода чисел от 0 до $n - 1$.

```
17 // числа от 0 до n - 1
18 for (int i = 0; i < n; i++)
19 {
20     Console.Write(i + " ");
21 }
```

Детали цикла:

- переменная-счетчик i инициализируется значением 0;
- цикл повторяется, пока $i < n$, т.е. для $i == n$ цикл уже не выполнится;
- $i++$ - значение i будет увеличиваться на 1 после каждой итерации;
- в консоль выводим в одну строку, поэтому после i пишем пробел и используем `Write`, а не `WriteLine`.

Добавим перед циклом вывод пояснения для пользователя. После цикла нужно перейти на новую строку в консоли.

```
17 // числа от 0 до n - 1
18 Console.WriteLine("Числа от 0 до (n-1):");
19 for (int i = 0; i < n; i++)
20 {
21     Console.Write(i + " ");
22 }
23 Console.WriteLine();
```

Проверьте работу программы (рис. 2.10). Установите breakpoint на строку с `for`. Запустите программу и, когда она остановится, продолжите выполнение в пошаговом режиме `F11`. Посмотрите, как изменяются значения переменных и содержимое консоли. После тестирования убедите этот breakpoint.

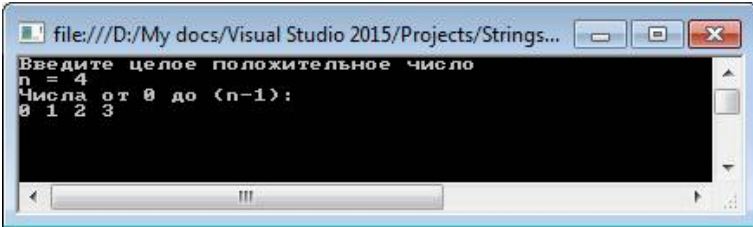


Рис. 2.10. Тестирование вывода значений в цикле

Числа от -n до n

Добавим еще один цикл. Он будет очень похож на предыдущий, поэтому можно просто скопировать его и отредактировать.

```
25 // числа от -n до n-  
26 Console.WriteLine("Числа от -n до n:");  
27 for (int i = -n; i <= n; i++)  
28 {  
29     Console.Write(i + " ");  
30 }  
31 Console.WriteLine();
```

Изменились только начальное и конечное значения. Теперь переменная `i` инициализируется значением `-n`, а в условии записано `i <= n`. То есть при `i == n` цикл еще выполнится, а в следующем значении `i == n + 1` остановится (рис. 2.11).

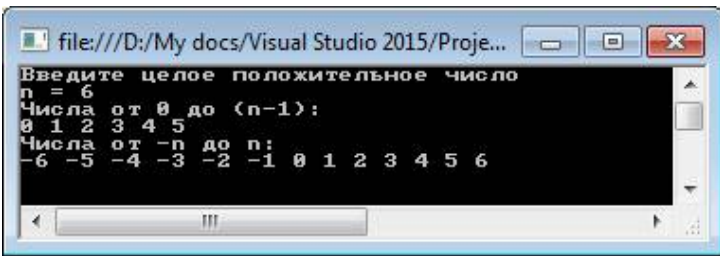


Рис. 2.11. Тестирование вывода значений от -n до n

Примечание. Счетчик *i* в двух созданных нами циклах - это разные переменные, хоть они и называются одинаково. Каждая "живет" только внутри своего цикла и удаляется из памяти после его завершения. И объявление мы делаем каждый раз отдельно (попробуйте удалить **int** во втором цикле - получите ошибку, хотя кажется, что такая переменная уже есть).

Можно задать и другое имя переменной-счетчика, это не принципиально.

Дроби от 1/n до 1/1

Цикл с перебором дробей будет отличаться по следующим пунктам:

- перебирать будем значения от *n* до 1, т.е. в обратном порядке: начальное значение *n*, условие *i* >= 1 (или *i* > 0), шаг *i*--;
- выводить будем значения дробей 1.0 / *i* (*i* - целочисленная переменная, поэтому если написать 1, а не 1.0, то деление будет нацело);
- каждую дробь будем выводить в отдельную строку, используем `WriteLine` с форматированием.

```
33 // числа от 1/n до 1/1
34 Console.WriteLine("Числа от 1/n до 1/1:");
35 for (int i = n; i >= 1; i--)
36 {
37     Console.WriteLine("1 / {0} = {1}", i, 1.0 / i);
38 }
```

Результат показан на рис. 2.12.

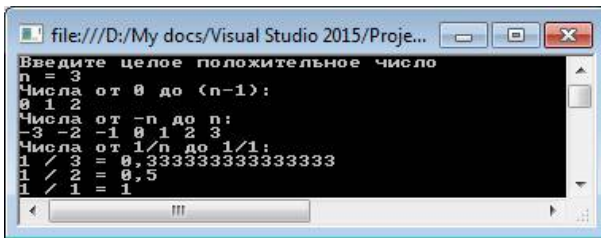


Рис. 2.12. Тестирование вывода значений от 1/n до 1/1

Числа, кратные *n*

Формулировка задания может звучать непонятно, но на самом деле оно очень простое. Числа, кратные *n*, т.е. делящиеся на *n* без остатка - это 1*n, 2*n, 3*n, 4*n и т.д.

Например, для *n* = 4: 1*4=4; 2*4 = 8; 3*4 = 12; 4*4 = 16; 5*4 = 20; ...

Принципиально изменился диапазон перебираемых значений. Теперь нам нужно вывести ровно 10 чисел, независимо от величины n , т.е. i будет изменяться от 1 до 10, а вывести нужно $i * n$.

Выводить будем опять в одну строку. Умножение выполним прямо внутри Write.

```
40 // 10 чисел, кратных n
41 Console.WriteLine("Числа, кратные n:");
42 for (int i = 1; i <= 10; i++)
43 {
44     Console.Write((i * n) + " ");
45 }
46 Console.WriteLine();
```

Результат показан на рис. 2.13.

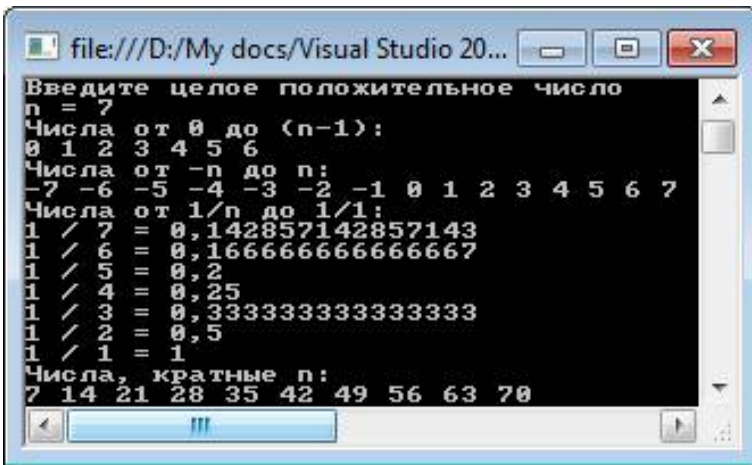


Рис. 2.13. Тестирование вывода значений, кратных n

Степени 2

Теперь нам нужно вывести степени числа 2 такие, чтобы они были меньше или равны n . Начинаем с $2^0 = 1$. Степень по-английски "power", поэтому в этот раз назовем переменную-счетчик p .

Каждая следующая степень 2 в два раза больше предыдущей, т.е. на каждом шаге нужно выполнить умножение счетчика на 2.

Условие останова цикла $p \leq n$. При этом необязательно цикл остановится точно в n . Главное, что когда p станет больше n , тогда цикл прервется.

```

48 // степени 2, не превосходящие n
49 Console.WriteLine("Степени 2, не превышающие n:");
50 for (int p = 1; p <= n; p *= 2)
51 {
52     Console.Write(p + " ");
53 }
54 Console.WriteLine();

```

Примеры для $n = 8$ и для $n = 70$ представлены на рис. 2.14 и 2.15.

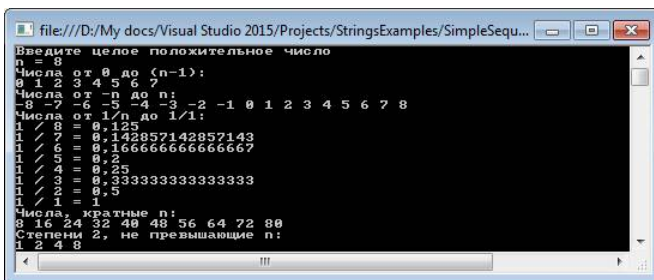


Рис. 2.14. Тестирование вывода степеней 2 от 1 до 8

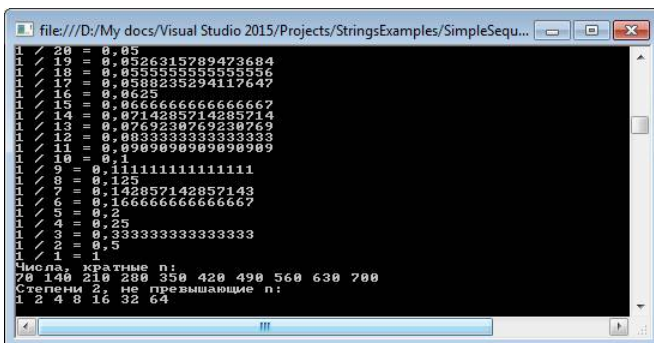


Рис. 2.15. Тестирование вывода степеней 2 от 1 до 70

Вложенные циклы (лесенка)

Последний этап задания - числовая лесенка. Здесь одним циклом мы не обойдемся: нам нужно перебирать "ступеньки", а внутри ступенек - числа от номера ступеньки до 0.

Таким образом, нам потребуется два **вложенных цикла** (аналог - матришка).

Первый, внешний, цикл будет перебирать "ступеньки" (строки) от 0 до n . Причем в конце ступеньки нужно будет начать новую строку, поэтому в конец тела цикла сразу вставим `WriteLine()`.

```

56 // числовая лесенка от n до 0
57 Console.WriteLine("Числовая лесенка:");
58 for (int i = 0; i <= n; i++)
59 {
60
61     Console.WriteLine();
62 }

```

Внутри первого цикла разместим второй, внутренний, цикл. У него должна быть другая переменная-счетчик, так как области действия циклов пересекаются. Назовем ее *j*. Значения *j* начинаются с номера ступеньки (*i*) и заканчиваются в 0, значения уменьшаются. Значения *j* выводим в одну строку.

```

56 // числовая лесенка от n до 0
57 Console.WriteLine("Числовая лесенка:");
58 for (int i = 0; i <= n; i++)
59 {
60     for (int j = i; j >= 0; j--)
61     {
62         Console.Write(j + " ");
63     }
64     Console.WriteLine();
65 }

```

Результат для $n = 9$ показан на рис. 2.16.

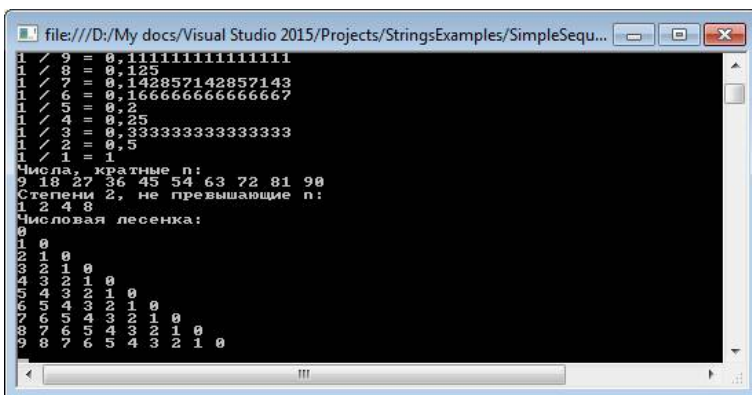


Рис. 2.16. Тестирование вывода числовой лесенки

? Что будет, если начальное значение *j* задать равным *n*?

Однако наша работа еще не закончена. В условии сказано, что если $n > 9$, то ступеньки больше 9 выводить не нужно.

Один из вариантов решения - если $n > 9$, то присвоить ему 9. Тогда цикл менять не придется.

```
if (n > 9)
{
    n = 9;
}
```

Но портить исходные данные нехорошо. Вместо этого добавим тернарный оператор?: во внешний цикл.

```
57 Console.WriteLine( "числовая лесенка: ");
58 for (int i = 0; i <= ((n > 9) ? 9 : n); i++)
59 {
```

Не забудьте поставить скобки вокруг выбора между n и 9. Вложенный цикл не изменится, он все равно начинается с i .

Проверим работу программы на $n = 40$ (рис. 2.17).

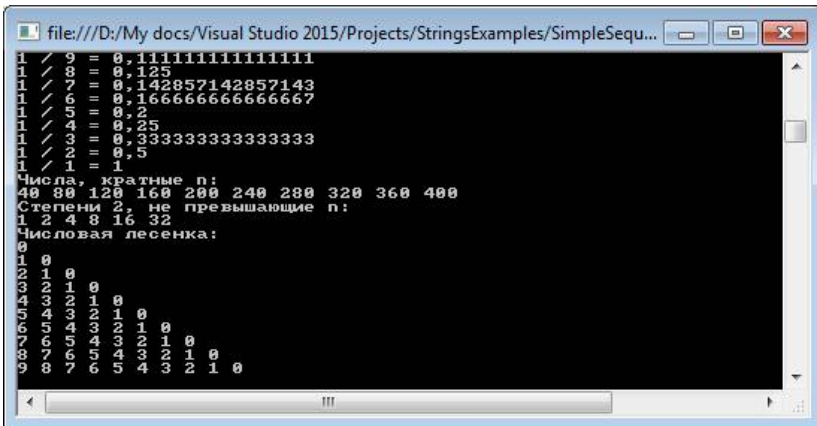


Рис. 2.17. Тестирование вывода числовой лесенки для чисел больше 9

Пример 3. Средние значения произвольной последовательности

Пользователь вводит целые числа по одному. Количество чисел заранее неизвестно. Ввод завершается, если пользователь ввел пустую строку. Если пользователь ввел строку, которая не является числом, то ее следует пропустить.

Вычислить среднее арифметическое, среднее геометрическое и среднее квадратическое введенных значений.

Среднее арифметическое - это сумма всех значений, деленная на их количество:

$$\frac{x_1 + x_2 + \dots + x_n}{n}$$

Среднее геометрическое- это произведение всех значений в степени 1/n (т.е. корень n-ной степени):

$$(x_1 \times x_2 \times \dots \times x_n)^{1/n}$$

Среднее квадратическое - это квадратный корень из суммы квадратов всех значений, деленной на их количество:

$$\sqrt{\frac{x_1^2 + x_2^2 + \dots + x_n^2}{n}}$$

Указания к выполнению

Создайте новый проект "Averages" в составе нашего решения "StringsAndArrays". Установите breakpoint, чтобы программа вставала на паузу, прежде чем закрыться.

Не забудьте переключить активный проект.

Последовательность произвольной длины

Теперь мы не знаем заранее, сколько раз будет повторяться цикл, поэтому **for** здесь не подходит.

While или do-while? Давайте подумаем. Мы будем вводить числа (много раз - значит внутри тела цикла). Если вместо числа введена пустая строка, то цикл останавливается. Значит, сначала должно быть тело цикла, а потом проверка условия. Это цикл с постусловием.

Введенное значение будем сравнивать с пустой строкой, значит, это тип `string`. Дадим переменной имя `input` (ввод). Объявить переменную нужно вне цикла, иначе она будет недоступна в условии.

```
11  > class Program {
12  {
13      Console.WriteLine("Введите последовательность чисел. Чтобы завершить ввод, нажмите Enter");
14      string input;
15
16      do
17      {
18          input = Console.ReadLine();
19      } while (input != "");
20  }
```

Проверьте работу программы: можно вводить любые числа и текст, как только ввели пустую строку, программа завершится.

Введенные значения нужно преобразовать в числовой тип. Поскольку сказано, что пользователь может ввести неверные значения и их

следует проигнорировать (вылета быть не должно), воспользуемся методом TryParse.

Значения могут быть дробными, поэтому выберем тип **double**.

```
string input;
double x;

do
{
    input = Console.ReadLine();
    double.TryParse(input, out x);
} while (input != "");
```

Количество, сумма и произведение в цикле

Добавим обработку введенных значений. Чтобы вычислить средние, требуется вычислить количество значений, их сумму и произведение.

Добавим переменные n, sum, prod (произведение - production) и sumSq (sum of squares, сумма квадратов). n - целое, а sum, sumSq и prod - дробные. Суммы и количество инициализируем нулем, а произведение 1.

```
15 string input,
16 double x;
17 int n = 0;
18 double sum = 0, prod = 1, sumSq = 0;
```

Примечание. Мы объявили несколько переменных одного типа в одну строку через запятую. Это допустимо, но нежелательно.

? Почему мы инициализировали произведение значением 1, а не 0?

В цикле если TryParse выполнен успешно, то n увеличить на 1, к sum прибавить x, prod умножить на x, а к sumSq прибавить квадрат x.

```
17 double sum = 0, prod = 1, sumSq = 0;

18
19
20 do
21 {
22     input = Console.ReadLine();
23     if (double.TryParse(input, out x))
24     {
25         n++;
26         sum += x;
27         prod *= x;
28         sumSq += x * x;
29     }
30 } while (input != "");
```

Мы еще никуда не выводим результат, но с помощью точки останова уже можем его проверить (рис. 2.18).

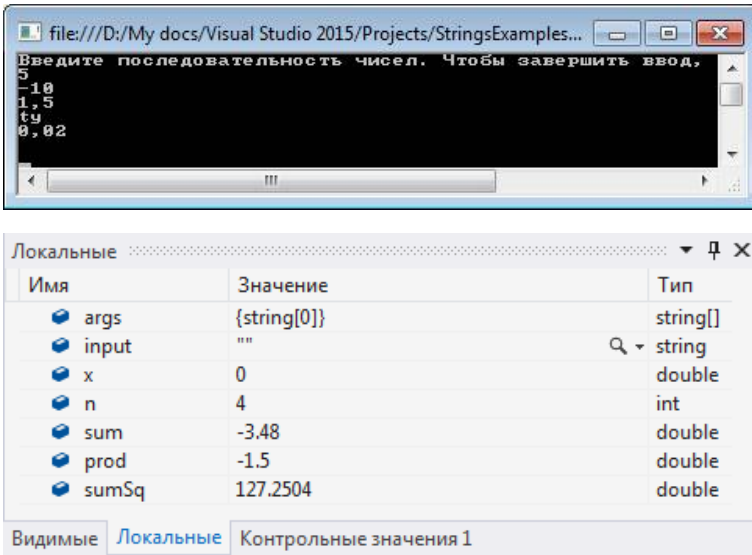


Рис. 2.18. Тестирование работы "Averages" с помощью breakpoint

При вводе пустой строки TryParse не сработал, поэтому $x = 0$. Количество введенных чисел $n = 4$ (текст "ty" не считается). Сумма:

$$5 + -10 + 1,5 + 0,02 = -3,48$$

Произведение:

$$5 \times -10 \times 1,5 \times 0,02 = -1,5$$

Сумма квадратов:

$$5 \times 5 + -10 \times -10 + 1,5 \times 1,5 + 0,02 \times 0,02 = 25 + 100 + 2,25 + 0,0004 = 127,2504$$

Как видим, все расчеты верны.

Добавим расчет средних и их вывод в консоль.

```
double arithmeticMean = sum / n;
double geomMean = Math.Pow(prod, 1.0 / n);
double sqMean = Math.Sqrt(sumSq / n);

Console.WriteLine("Среднее арифметическое: {0}", arithmeticMean);
Console.WriteLine("Среднее геометрическое: {0}", geomMean);
Console.WriteLine("Среднее квадратическое: {0}", sqMean);
```

? Почему в вычислении geomMean нужно писать 1.0/n, а не 1/n? Проверка работы программы (рис. 2.19).

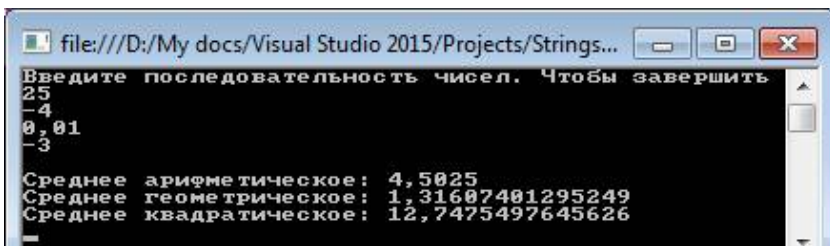


Рис. 2.19. Тестирование работы Averages

Через предусловие

Хотя мы изначально определили тип цикла - с предусловием - синтаксис, C# достаточно гибок, чтобы задачу можно было решить и через while.

Можно разместить присваивание input прямо в условии цикла. Тогда мы сначала вводим значение, проверяем его и, если это не пустая строка, переходим к расчетам.

```

while ((input = Console.ReadLine()) != "")
{
    if (double.TryParse(input, out x))
    {
        n++;
        sum += x;
        prod *= x;
        sumSq += x * x;
    }
}

```

Примечание. Объявлять переменные в условии нельзя.

Еще один вариант, более классический: первое значение ввести при объявлении переменной, а следующие вводить в конце тела цикла.

```

string input = Console.ReadLine();
double x;
int n = 0;
double sum = 0, prod = 1, sumSq = 0;

while (input != "")
{
    if (double.TryParse(input, out x))
    {
        n++;
        sum += x;
        prod *= x;
        sumSq += x * x;
    }
    input = Console.ReadLine();
}

```

Протестируйте работу программы с различными типами циклов.

Пример 4. Обработка массива

Пользователь вводит массив x в виде строки с числами, разделенными пробелами. Числа могут быть дробными и отрицательными.

1. Выведите значения массива x в обратном порядке, в строку через точку с запятой.

2. Создайте копию массива. Отсортируйте эту копию по возрастанию. Выведите в четыре столбца: i - номера элементов, x - значения исходного массива, $xSort$ - значения отсортированного массива, $rank$ - ранг каждого значения массива x (порядковый номер значения в отсортированном массиве, начиная с 1). Используйте табуляцию для выравнивания столбцов.

3. Выведите минимальное и максимальное значения, сумму и среднее значение массива.

4. Выведите медиану массива (значение, которое стоит в середине отсортированного массива).

Пример:

```
Введите числа через пробел:
23 7 30 -14 37 21 44 28 51 0 58 42 65 49
В обратном порядке:
49; 65; 42; 58; 0; 51; 28; 44; 21; 37; -14; 30; 7;
23

Сортировка:
i      x      xSort rank
0      49     -14    11
1      65      0     14
2      42      7      9
3      58     21     13
4       0     23      2
5      51     28     12
6      28     30      6
7      44     37     10
8      21     42      4
9      37     44      8
10     -14     49      1
11     30     51      7
12      7     58      3
13     23     65      5

Минимальное: -14
Максимальное: 65
Сумма: 441
Среднее: 31,5
Медиана: 30
```

Массивы

Массив - это последовательность однотипных значений. Массив является структурой, т.е. сложным типом данных.

Может быть массив целых чисел, массив дробных чисел, массив дат, массив строк и т.д. Но не могут быть в одном массиве элементы разных типов. Примеры числового и строкового массивов приведены в табл. 2.2 и 2.3.

Таблица 2.2

Пример числового массива

Индекс	0	1	2	3	4	5	6	7	8
Элемент	5	100	-7	0	10	12	15	20	25

Таблица 2.3

Пример строкового массива

Индекс	0	1	2	3	4	5
Элемент	"На"	"дворе"	"траве"	"на"	"траве"	"дрова"

Длина массива (т.е. число значений в нем) может быть заранее неизвестна. Но после создания массива его длину изменить уже нельзя.

Каждый **элемент массива** имеет номер (**индекс**), по которому его можно получить. Индексы отсчитываются с нуля.

Индекс пишется в квадратных скобках после имени массива. Например, `a[5]` - шестой элемент в массиве `a` (потому что отсчет начинается с 0).

Попытка обратиться к несуществующему индексу вызовет "вылет" программы с ошибкой `AccessViolation` (нарушение доступа).

Массив **объявляется** через любой другой тип, который будут иметь его элементы. После типа данных ставятся пустые квадратные скобки.

Примеры:

```
int[] x; // целые числа
double[] weights; // дробные числа
decimal[] wages; // числа с десятичной точкой
string[] usernames; // строки
```

Имена переменных массивов желательно писать во множественном числе.

Хранимые и ссылочные типы

Все простые типы данных, которые мы рассматривали до сих пор, были хранимыми. Массивы - это ссылочные типы данных.

Хранимые (размерные, значимые, value) типы данных имеют небольшую постоянную длину. Место в памяти под такие переменные выделяется сразу при запуске программы. Например, тип `long` - под каждую переменную будет выделено 8 байт, и ее значение всегда будет храниться в этих восьми байтах. Переменная типа `int` займет 4 байта, и она будет храниться в этих четырех байтах, пока не закончится ее область действия. Даже если переменная не будет использоваться, место в памяти все равно зарезервировано.

Ссылочные (reference) типы данных - объекты, которые имеют большую и/или переменную длину (массивы, строки и др.). Сколько им надо памяти - заранее может быть неясно (в массиве может быть один элемент, а может миллион, строка может быть пустой, а может занимать до 2 ГБ).

В данном случае в переменной сохраняют ссылку (reference) на адрес в динамически распределяемой памяти, где уже отводится место под значение - столько, сколько нужно. Длину значения можно в любой момент изменить, а можно вообще удалить объект, когда он уже не нужен.

Переменные ссылочных типов нужно не только объявлять, но и создавать (инициализировать) командой **new** либо начальным значением.

Пока ссылочная переменная не инициализирована, она равна особому значению **null**.

Инициализация массивов

Способ 1 (просто, но применяется редко): сразу ввести все значения в фигурных скобках. Значения должны быть известны заранее.

```
int[] a = {5, -2, 1, 6, 9};  
string[] weekdays = { "Пн", "Вт", "Ср", "Чт", "Пт",  
                      "Сб", "Вс" };
```

Способ 2: выделить в памяти место под массив командой **new**. Длину массива можно задавать через переменную, значения заранее неизвестны. Массив инициализируется нулями или пустыми строками.

```
int[] a = new int[5];  
decimal[] wages = new decimal[n];  
string[] names = new string[userCount];
```

Попытка обратиться к неинициализированному массиву, как и неверное значение индекса, вызывает исключение `AccessViolation` и "вылет" программы.

Стандартные методы массивов

Массивы, как и все в C#, имеют свой *класс* - это системный **класс Array**. Он предоставляет методы для создания, изменения, поиска и сортировки.

Свойства массива:

- `arr.Length` - длина массива `arr`.

Статические методы класса:

- `Array.Copy(src, dst, n)` - скопировать из массива `src` в массив `dst` первые `n` значений;
- `Array.Reverse(arr)` - переворачивает массив `arr` на месте (обратный порядок значений);
- `Array.Sort(arr)` - сортирует массив `arr` по возрастанию;
- `Array.IndexOf(arr, value)` - ищет в массиве `arr` значение `value` и возвращает номер его первого вхождения, а если значение не найдено, то `-1`.

Кроме того, массив поддерживает интерфейс **списка (List)**, который предоставляет свои методы.

Методы списков:

- `arr.Min()`, `arr.Max()` - минимальное и максимальное значения в массиве `arr`;
- `arr.First()`, `arr.Last()` - первое и последнее значения в массиве `arr`;
- `arr.Sum()` - сумма значений в массиве `arr` (только для числовых);
- `arr.Average()` - среднее значение в массиве `arr` (только для числовых).

Указания к выполнению

Создайте новое консольное приложение с именем "ArrayMethods" в том же решении и сделайте его активным. Установите `breakpoint` в конце программы.

Преобразование строки в массив

По заданию все числа вводятся в одну строку через пробел, а не в отдельных строках, как мы делали в предыдущих примерах.

У типа `string` существует встроенный метод `Split`, который позволяет автоматически разбить строку по заданному символу (пробел или любой другой) и получить массив значений.

```

11     static void main(string[] args)
12     {
13         Console.WriteLine("Введите числа через пробел:");
14         string[] strArray = Console.ReadLine().Split(' ');

```

? Почему пробел в методе `Split` записан в одинарных, а не в двойных кавычках?

Правда, в результате получим массив строк, а нам нужен массив чисел. Поэтому потребуется два массива разных типов.

Числовой массив по длине совпадает со строковым. В цикле `for` мы проходим по всем значениям и каждое преобразуем в тип `double`.

```

14+         string[] strArray = Console.ReadLine().Split(' ');
15
16         int n = strArray.Length;
17         double[] arr = new double[n];
18         for (int i = 0; i < n; i++)
19         {
20             arr[i] = double.Parse(strArray[i]);
21         }
22
23

```

Обратите внимание на границы индекса `i`: начинаем в 0, а заканчиваем, не доходя до `n` (строго меньше).

Преобразование массива в строку

В первом пункте задания требуется развернуть массив. Это действие делается в одну строку с помощью встроеного метода: `Array.Reverse(arr)`.

При этом массив `arr` перевернется "на месте", т.е. будут переставлены местами его элементы и он изменится.

Для вывода результата в строку используем метод `Join` типа `string` с разделителем точка с запятой и пробел.

В конце разворачиваем массив еще раз, чтобы вернуть его в первоначальное состояние.

```

23         Console.WriteLine("В обратном порядке:");
24         Array.Reverse(arr);
25         Console.WriteLine(string.Join("; ", arr));
26         Array.Reverse(arr);

```

Протестируйте работу программы на данном этапе (пример на рис. 2.20).

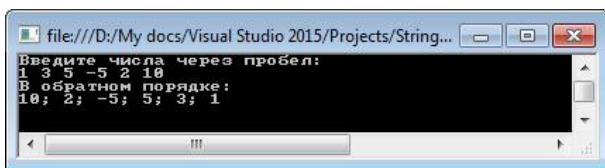


Рис. 2.20. Тестирование разворота массива в "ArrayMethods"

Копирование и сортировка массива

Сортировка массива также выполняется встроенным методом `Sort`, и он также изменяет массив, к которому применяется. Поэтому создадим копию исходного массива.

Напомним, что массивы являются ссылочными типами. А это значит, что если мы просто присвоим новой переменной значение `arr`, то массив останется один, просто будет две переменных, которые ссылаются на него:

```
double[] arrCopy = arr; //новый массив не будет создан
```

При такой записи массив останется один, а `arr` и `arrCopy` - это просто два разных имени этого массива.

Чтобы создать полноценную копию массива, нужно использовать метод `Array.Copy` (откуда, куда, сколько). Перед копированием оба массива должны быть инициализированы. После этого копию можно отсортировать.

```
28     double[] arrSort = new double[n];
29     Array.Copy(arr, arrSort, n);
30     Array.Sort(arrSort);
```

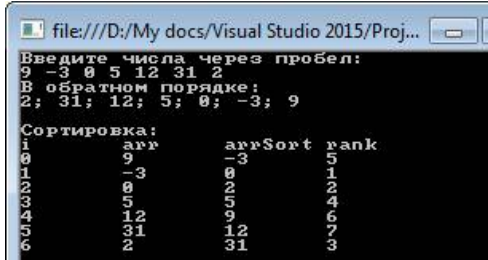
Далее выведем таблицу со значениями исходного и отсортированного массива, а также с рангами исходных значений.

Ранг (rank) - это порядковый номер значения в отсортированном массиве, т.е. нам нужно определить индекс (метод `IndexOf`) каждого значения из массива `arr` в `arrCopy`. Учтем, что ранги отсчитываются с 1, а не с 0, как индексы массивов.

```
28     Console.WriteLine("\nСортировка:");
29     double[] arrSort = new double[n];
30     Array.Copy(arr, arrSort, n);
31     Array.Sort(arrSort);
32     Console.WriteLine("i\tarr\tarrSort\trank");
33     for (int i = 0; i < n; i++)
34     {
35         int rank = Array.IndexOf(arrSort, arr[i]) + 1;
36         Console.WriteLine("{0}\t{1}\t{2}\t{3}", i, arr[i], arrSort[i], rank);
37     }
```

Для форматирования выходной строки мы использовали управляющие последовательности (символы) `\n` (переход на новую строку) и `\t` (табуляция).

Результат показан на рис. 2.21.



```
file:///D:/My docs/Visual Studio 2015/Proj...
Введите числа через пробел:
9 -3 0 5 12 31 2
В обратном порядке:
2; 31; 12; 5; 0; -3; 9
Сортировка:
i      arr      arrSort  rank
0      9          -3        5
1     -3         0        1
2      0         5        2
3      5         9        4
4     12         9        6
5     31        12        3
6      2         31        7
```

Рис. 2.21. Тестирование сортировки массива в "ArrayMethods"

Методы числовых массивов

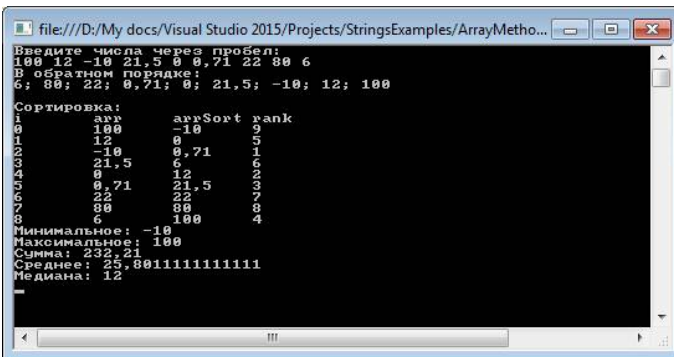
Осталось вывести характеристики массива: минимальное и максимальное значения, сумму и среднее значение массива, медиану массива. Все их можно получить из стандартных методов, а медиану - из середины отсортированного массива.

Ранее рассмотренные методы Reverse, Sort, IndexOf были статическими, т.е. вызывались для самого класса Array, а переменная массива в них передавалась как параметр.

Методы Min, Max, Sum, Average - это обычные, нестатические методы, и вызываются они из самой переменной массива.

```
36
39 Console.WriteLine("Минимальное: {0}", arr.Min());
40 Console.WriteLine("Максимальное: {0}", arr.Max());
41 Console.WriteLine("Сумма: {0}", arr.Sum());
42 Console.WriteLine("Среднее: {0}", arr.Average());
43 Console.WriteLine("Медиана: {0}", arrSort[n / 2]);
44
```

Результат показан на рис. 2.22.



```
file:///D:/My docs/Visual Studio 2015/Projects/StringsExamples/ArrayMetho...
Введите числа через пробел:
100 12 -10 21,5 0 0,71 22 80 6
В обратном порядке:
6; 80; 22; 0,71; 0; 21,5; -10; 12; 100
Сортировка:
i      arr      arrSort  rank
0     100     -10        9
1     12         0        5
2    -10     0,71       1
3    21,5     12        3
4      0      12        2
5     0,71    21,5       7
6     22      22        4
7     80      80        6
8     100     100        8
Минимальное: -10
Максимальное: 100
Сумма: 232,21
Среднее: 25,801111111111111
Медиана: 12
```

Рис. 2.22. Тестирование методов числовых массивов в "ArrayMethods"

Задачи для самостоятельного решения

Задача 2.1. Округление

Класс `Math` включает в себя целых 4 функции округления: `Math.Ceiling`, `Math.Floor`, `Math.Round`, `Math.Truncate`. Выясните, чем они отличаются.

Для этого напишите программу, в которой константой будет задан массив значений `[3.123, 12.5, 8.76, -4.3, -100.5, -9.99999]`. В цикле переберите эти значения и выведите результат округления каждой функцией. Сделайте выводы. Какой тип имеет результат округления?

Задача 2.2. Сортировка по алфавиту

Пользователь вводит строку со словами, записанными через пробел. Необходимо отсортировать эти слова по алфавиту.

Например, входная строка: “Семен абажур чемодан Чебурашка мел кольцо”.

Выходная строка: “абажур кольцо мел Семен Чебурашка чемодан”.

Используйте методы `string.Split`, `string.Join` и `Array.Sort`.

Задача 2.3. Коды Unicode

Напишите программу, которая для любой нажатой клавиши выводит ее код в таблице Unicode.

Зациклите работу программы, чтобы она запрашивала символы бесконечно, пока не будет закрыта.

Задача 2.4. Кассовый чек v.0.1.1

Дополните задачу 1.3 так, чтобы пользователь мог ввести несколько товаров: сначала запросите количество товаров, а затем их цены и количество в цикле. Сразу выводите стоимость каждого товара, а в конце покажите общую сумму. В данной версии программы нет необходимости использовать массивы.

Пример:

```
Введите число наименований товаров, а затем цену и стоимость каждого товара.
```

```
Число товаров: 3
```

```
Товар 1:
```

```
Цена: 2,5
```

```
Кол-во: 2
```

```
Стоимость: 5
```

```
Товар 2:
```

```
Цена:      45
Кол-во:    1
Стоимость: 45
```

```
Товар 3:
Цена:      100
Кол-во:    5
Стоимость: 500
```

```
==
Сумма:     550,00
```

Введите полученную сумму наличных.

```
Получено: 600,00
Сдача:    50,00
```

Задача 2.5. Кассовый чек v.0.1.2

Добавьте в предыдущую задачу возможность вывести итоговый чек в отформатированном виде. Самостоятельно изучите, как можно отформатировать составные строки.

1. Сохраняйте все количества и цены товаров в массивы.

2. Добавьте ввод названия товара.

3. После того, как все данные введены, очистите консоль командой `Console.Clear()` и выведите отформатированный чек в соответствии с примером. Ширину отступов подберите самостоятельно.

4. В начале чека выведите текущую дату и время в формате ДД.ММ.ГГГГ ЧЧ:ММ:СС.

5. Вычислите и выведите долю стоимости каждого товара в общей сумме чека в процентах.

Пример чека:

```
04.11.2018 11:05:12
```

```
Пакет
```

```
2,50 * 2 = 5,00 (2,5%)
```

```
Молоко
```

```
45,00 * 1 = 45,00 (21,5%)
```

```
Яблоки
```

```
129,95 * 1,224 = 159,60 (76,1%)
```

```
=====
```

```
ИТОГ: 209,06 (100%)
```

```
Получено: 500,00
```

```
Сдача: 290,94
```

Контрольные вопросы

1. Чем отличаются символьный и строковый тип данных?
2. Что такое пустая строка?
3. Каким образом сравниваются строки?
4. Как выделить подстроку в строке?
5. Как можно изменить регистр символов в строке?
6. Какие циклы используются в языке C#?
7. Запишите цикл `for` для вычисления суммы, произведения и подсчета количества значений.
8. Что такое вложенные циклы?
9. Что такое массив? Как объявляются переменные массивов?
10. Запишите цикл `for` для перебора элементов массива.
11. Какие вы знаете встроенные методы для работы с массивами?
12. Что делает метод `Split`?
13. Что такое проект и решение в VisualStudio?

3. ООП. КЛАССЫ, АТТРИБУТЫ, МЕТОДЫ. СПИСКИ

Основные понятия ООП

C# - это объектно-ориентированный язык программирования. Понятия класса и объекта являются для него базовыми, без их понимания невозможно вести никакую разработку.

Класс - группа реальных или виртуальных объектов, обладающих общими свойствами (аттрибутами) и методами поведения.

Объект (экземпляр класса, инстанция) - конкретный представитель класса с определенными значениями свойств.

Примеры приведены в табл. 3.1.

Таблица 3.1

Примеры классов и экземпляров

Класс	Экземпляр
Кошка	Конкретная Мурка
Студент	Конкретный студент Н. Кузнецов
Чек	Чек №0001003234023 от 02.11.2012
Командная строка	Конкретное окно командной строки, которое появилось при запуске программы

В зависимости от поставленной задачи граница между экземпляром и классом может смещаться. Например, класс - автомобиль. Если задача просто описать различные марки автомобилей, то экземпляром может быть "LADA Granta" или "Ford Focus" (не какой-то один реальный автомобиль, а в целом все автомобили этой марки). Но если задача - составление объявлений о продаже подержанных авто, то в этом случае экземпляром будет вполне конкретный автомобиль с конкретным владельцем, а его марка всего лишь атрибут.

Атрибут (свойство) - это некоторая характеристика, присущая всем объектам данного класса. Ее значения могут отличаться, но у всех она присутствует.

Можно сказать, что объект отличается от класса тем, что у объекта атрибуты имеют конкретные значения, а у класса просто перечисляются (описываются).

Примеры атрибутов классов и их значений для конкретных экземпляров представлены в табл. 3.2.

Таблица 3.2

Примеры атрибутов классов и их значений у экземпляров

Класс	Атрибуты	Экземпляр
Кошка	Имя Пол Возраст Окрас ТипШерсти	Мурка жен. 3 Черный с белыми "носочками" Короткая
Студент	ФИО ДатаРождения НомерСтудБилета Группа Курс	Н.А. Кузнецов 15.06.1998 17100187 ПИ 1
Чек	№ чека ДатаВремяВыдачи Кассир ПереченьТоваров Сумма	0001003234023 02.11.2012 15:04 Добрынина (сами товары - тоже объекты) 312,15
Командная строка	КоординатаX КоординатаY Ширина Высота ЦветТекста ОтображаемыйТекст	100 315 640 480 White "Введите число:?"

Имена атрибутов принято писать в PascalStyle - каждое слово с заглавной буквы без пробелов и знаков подчеркивания.

При описании атрибутов принято сразу указывать **тип данных**. При этом обычно не привязываются к конкретному языку программирования, а просто используют соответствующее по смыслу слово на английском или родном языке автора (Boolean - Логический, Number - Числовой, Integer - Целый, Float - С плавающей запятой, Currency - Финансовый, Text - Текстовый, String - Строка и др.). Можно указать и до-

полнительные характеристики типа: например, длину строки или массива в квадратных скобках или условия назначения (табл. 3.3).

Таблица 3.3

Примеры типов данных атрибутов

Класс	Атрибуты	Тип данных (рус.)	Тип данных (англ.)
Кошка	Имя Пол Возраст Окрас ТипШерсти	Строка Логический Целое ≥ 0 Строка Строка	String Boolean Integer ≥ 0 String String
Студент	ФИО ДатаРождения НомерСтудБилета Группа Курс	Строка Дата Строка[8] Строка Целое > 0	String Date String[8] String Integer > 0
Чек	НомерЧека ДатаВремяВыдачи Кассир ПереченьТоваров Сумма	Строка[15] ДатаВремя Строка Массив товаров Финансовый	String[15] DateTime String Item[] Currency
Командная строка	КоординатаX КоординатаY Ширина Высота ЦветТекста ОтображаемыйТекст	Целое Целое Целое > 0 Целое > 0 Цвет консоли Массив строк	Integer Integer Integer > 0 Integer > 0 ConsoleColor String[]

Метод (операция) - определенный способ поведения (действие), присущий всем экземплярам класса. Действие может быть как активным (сам объект что-то делает), так и пассивным (с объектом что-то делают).

Метод задается для всего класса и действует для всех его объектов (примеры в табл. 3.4).

Перечень методов опять же зависит от задачи, ради которой мы описываем класс. В примере для класса "Студент" мы описали только методы поведения, связанные с учебой в вузе, хотя студент может выполнять

еще массу других действий. Но, например, метод "Играть на саксофоне" определенно не подходит, так как не все студенты это умеют.

После имени метода обязательно ставятся **скобки**, даже если они пустые. Пробел между именем метода и скобками не ставится. В скобках пишутся через запятую **параметры**, которые нужно передать методу извне, чтобы его выполнить.

Таблица 3.4

Примеры методов классов

Класс	Методы
Кошка	Мяукать() Мурлыкать() Спать() Играть() Есть(еда)
Студент	Учиться(лекция) Учиться(семинар) Отдыхать(длительность) Отдыхать(началоПеремены, конецПеремены) Прогуливать(пара)
Чек	Напечатать() ВычислитьСумму(): Финансовый ПроверитьСдачу() ВыделитьНДС()
Console	Write(string value) WriteLine(string value) ReadLine() ReadKey() Clear()

Например, кошке нужно передать еду, которую она съест. Студенту нужно предоставить лекцию или семинар, где он будет учиться. Консоли нужно передать текст, который она должна напечатать, но ничего не нужно передавать для ввода или очистки.

Чеку не нужно передавать список товаров, чтобы вычислить сумму, так как эти товары уже содержатся в самом чеке (см. атрибуты чека в табл. 3.2).

В совокупности атрибуты и методы называют членами (members) класса.

Представление классов в программах

С точки зрения программирования на любом языке **класс** - это **структура данных**, т.е. описание того, какие данные (типы) и в каком порядке нужно хранить в памяти компьютера (атрибуты) и какие программы можно над этими данными выполнять (методы). А **объект** - это уже **записанные в память данные** с заполненными значениями атрибутов.

Класс объявляется ключевым словом **class**, затем идет уникальное **имя класса** (в PascalStyle), затем в фигурных скобках `{ }` - **описание класса**:

```
class Cat //класс Кошка
{
    //здесь будет описание класса
}
```

Атрибуты в программировании могут быть двух основных типов: поля и свойства. Со свойствами мы разберемся немного позже, а **поля** являются аналогами переменных, объявленных внутри класса. Они также имеют тип и имя и могут иметь начальное значение.

```
class Cat //класс Кошка
{
    string name = ""; //имя
    byte age = 0; //возраст
    bool female = false; // пол
    double weight = 1; //вес
}
```

Методы являются подпрограммами (функциями) в теле класса.

Сначала указывается тип данных, который возвращает метод. Если метод не возвращает ничего, то пишется ключевое слово **void**. Далее идет имя метода в PascalStyle, затем в круглых скобках - перечисление параметров через запятую с указанием типов данных.

После объявления метода ставятся фигурные скобки и пишется его реализация. Между методами оставляют как минимум одну пустую строку, чтобы они не сливались визуально. Перед методом принято оставлять комментарий с пояснением, что этот метод делает и что означают его параметры.

```
//класс Кошка
class Cat
{
```



```

string name = ""; //имя
byte age = 0; //возраст
bool female = false; // пол
double weight = 1; //вес

// сказать текст speech
void Say(string speech)
{
    //speech выводится в консоль
    Console.WriteLine(speech);
}

// любит ли еду food?
bool Likes(string food)
{
    //любит мясо, рыбу и молоко
    if (food == "мясо" || food == "рыба" || food ==
        "молоко")
        return true;
    //все остальное - не любит
    return false;
}

// съесть еду food весом foodWeight
void Eat(string food, double foodWeight)
{
    //если любит, то съест
    if (Likes(food))
    {
        //и прибавит в весе
        weight += foodWeight;
    }
}
}

```

В то же время класс можно рассматривать как собственный (созданный программистом) тип данных. Можно (и нужно) объявить переменную типа `Cat` и **создать новый экземпляр класса**.

```
Cat murka = new Cat();
```

В данном случае `murka` - это экземпляр класса `Cat`, т.е. объект. Мы можем обращаться к полям и методам класса `Cat` с помощью знакомого нам оператора "точка":

```
murka.name = "Мурка";  
murka.age = 3;  
murka.female = true;  
murka.weight = 4.1;  
murka.Eat("рыба", 0.2);
```

Ссылочные и хранимые типы данных

Фактически все, что есть в `C#`, является классами и объектами.

Ранее мы уже использовали некоторые классы: `Console`, `Math`, `Program`. Вы уже знаете, что достаточно написать имя класса, поставить точку, и подсказка `IntelliSense` выведет список доступных атрибутов и методов (например, на рис. 3.1).

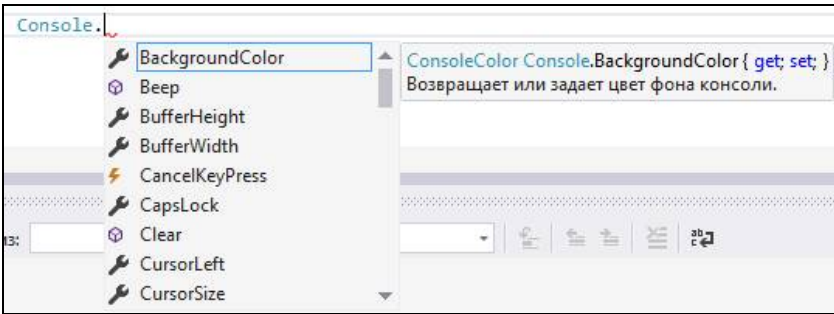


Рис. 3.1. Список методов и атрибутов класса `Console`

Даже простейшие типы, такие как `int`, `double`, `string`, имеют "класс-близнец", реализованный в `.NET`, соответственно, `Int`, `Double`, `String` (с заглавной буквы). Именно этот факт позволяет нам использовать конструкции вида:

```
int.TryParse(str, out x)  
str.Length
```

То есть на самом деле мы обращаемся к методам соответствующих классов.

Но между простейшим типом данных и классом есть существенная разница: класс является **ссылочным типом данных**, а `int`, `double`, `boolean` и т.д. - это **хранимые типы данных**. Это значит, что если мы объявляем переменную

```
int x = 5;
```

то в переменной `x` сохраняется само значение 5 "как есть", в двоичной форме.

Но когда мы объявляем:

```
Cat murka = new Cat();
```

то в самой переменной `murka` нет объекта "Кошка". Там размещается адрес в памяти (ссылка), где фактически находится этот объект.

Если мы объявим еще одну переменную и скопируем туда значение:

```
Cat murzik = murka;
```

то в переменной `murzik` будет копия **ссылки** на ту же самую кошку `murka`, а вовсе не новая кошка (правильнее было бы назвать переменную `murka2`, а не `murzik`).

Известные нам типы данных (а мы рассмотрели далеко не все) показаны на рис. 3.2.

Чтобы лучше понять, как это работает, рассмотрим аналогию.

Представьте себе, что распределяемая **память**, где находится запущенная программа и ее данные, - это такой склад, в котором все хранится на абсолютно одинаковых полках в одинаковых стандартных ящиках (это **ячейки памяти** размером 1 байт). Полки пронумерованы подряд (это **адрес** ячейки в памяти). Причем все полки выглядят абсолютно одинаково, с одинаковыми ящиками, независимо от того, что в этих ящиках лежит или они вообще пустые (все в **двоичном коде**).

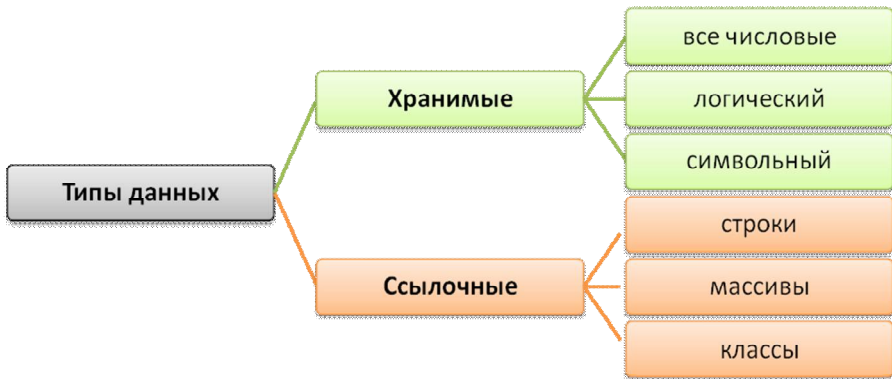


Рис. 3.2. Классификация типов данных на ссылочные и хранимые

Как найти какой-нибудь объект на этом огромном складе с миллиардами полок? Надо знать номер полки, на которой он лежит. **Ссылка на объект (указатель, reference)** - это как раз запись с номером полки (адресом памяти). По сути, указатель является просто целым числом, как и тип `int`, и сам по себе ничего не говорит о том, что лежит на пол-

ке. Так что одного указателя может быть недостаточно для работы, еще нужно знать, что именно находится по указанному адресу. За это отвечает тип переменной.

Объекты могут быть как больше, так и меньше стандартной полки (одного байта).

Если объект меньше полки, он все равно займет ее целиком, обычно нельзя хранить два объекта вместе в одном байте (тут могут быть исключения).

Если объект больше полки, то он будет занимать несколько полок подряд. Тогда надо еще знать, сколько места он занимает (**размер, длину**).

Даже одно целое число (тип `int`) занимает 4 байта, а массивы, строки и объекты - десятки и сотни байт.

Сам **класс** - это описание объектов, из чего они состоят и сколько места занимают (если знаем адрес начала объекта, то по размеру класса найдем, где он заканчивается). А в памяти хранятся **экземпляры класса** с заполненными значениями полей.

При создании нового экземпляра класса (или массива) программа "смотрит", сколько места в памяти занимает этот экземпляр (или сколько мы запросили места для массива). Далее следует найти в памяти нужное количество идущих подряд пустых ячеек. Если свободное место не будет найдено, можно получить ошибку доступа к памяти. Такое случается в "тяжелых" приложениях и при допущенных ошибках управления памятью в программе.

Как только место будет найдено, его необходимо **зарезервировать**, а в переменную записать **адрес начала объекта** (иначе он тут же потеряется). Поэтому команда выделения памяти `new` обязательно идет в связке с присваиванием.

Еще один важный момент - кто управляет этим складом? В первую очередь, операционная система (ОС). Она выделяет программам память для работы. Одна программа не должна свободно залезать в данные другой программы и тем более в системные области памяти - за этим тоже следит ОС. Своей выделенной областью памяти программа распоряжается сама (исходя из того, что в нее заложил программист).

Свою роль играет .NET и среда CLR, в которой выполняются все программы на C#.

Важная функция, которую выполняет CLR, - это **очистка памяти** от ненужных объектов. Среда постоянно анализирует программу и как только видит, что какой-то объект больше не нужен (нет переменных,

которые содержат его адрес, или эти переменные больше не используются), CLR этот объект удаляет и освобождает память.

Во многих других языках программирования программист должен сам удалить объект специальной командой. Если программист забудет это сделать, то объект останется висеть в памяти и будет занимать место. Когда таких объектов накапливается много, говорят об **утечках памяти**. В C# тоже можно написать достаточно кривой код, чтобы возникли утечки, но в большинстве случаев .NET спасает от этой проблемы.

Пример 1. Прототип игры "Fluffies"

Разработать консольный прототип игры "Fluffies" ("Пушистики").

В игре необходимо создать своего питомца и заботиться о нем. В прототипе доступна только кошка. Игра ведется пошагово, на каждом шаге пользователь может выбрать действие.

У кошки есть имя, цвет, возраст и показатели голода, усталости и счастья.

– Именем кошки может быть произвольная строка текста.

– Цвет в текущей версии выбирается из системных цветов консоли.

– Возраст в начале игры равен 0 и увеличивается на 1 после каждого хода.

– Показатели голода, усталости и счастья изменяются в диапазоне от 0 до 10. Пока что это целые числа, в будущем могут быть дробными. В начале игры все показатели равны 5.

Счастье зависит от голода и усталости. Если в конце хода голод меньше 3, то счастье увеличивается на 1. Если голод больше 5, то счастье снижается на 1, если больше 8, то на 2. Если усталость больше 5, то счастье снижается на 1, если больше 8, то на 2.

Кошка может ничего не делать, спать, с ней можно поиграть, ее можно покормить и приласкать.

– Когда кошка ничего не делает, ее голод увеличивается на 1, а счастье снижается на 1.

– Кошка уснет, если ее усталость больше 8. После сна усталость снижается до 0, а голод увеличивается на 4.

– Если кошку покормить, то ее голод падает до 0, а усталость и счастье увеличиваются на 1. Кормить можно только голодную кошку (голод больше 3).

– Если с кошкой поиграть, то ее счастье увеличивается на 3, голод увеличивается на 2 и усталость на 3. Если кошка голодна или устала (больше 8), то она откажется играть.

– Если кошку приласкать, то все показатели увеличатся на 1.

Изменение показателей представлено в виде табл. 3.5.

Таблица 3.5

Изменение показателей кошки в игре "Fluffies"

Действие \ Показатель	Ничего не делать	Спать	Есть	Играть	Ласкать
Голод	+1	+4	0	+2	+1
Усталость	+0	0	+1	+3	+1
Счастье	-1	+0	+1	+3	+1

На каждом шаге игры пользователь выбирает действие нажатием соответствующей клавиши. После этого состояние кошки обновляется и выводится на экран. Игра длится до бесконечности.

Проектирование класса "Кошка"

Для реализации игровой механики создадим класс "Кошка" ("Cat"). Данный пример отличается от того, что мы описывали в теории, - там не было привязки к конкретной задаче. В условии достаточно четко описаны атрибуты и методы класса. Давайте выпишем их с указанием типа данных (табл. 3.6).

Таблица 3.6

Члены класса "Кошка" на русском и английском языках

Класс Кошка	class Cat
<i>Атрибуты:</i>	<i>Attributes:</i>
Имя: Текст	Name: String
Возраст: Целое > 0	Age: Integer > 0
Цвет: ЦветКонсоли	Color: ConsoleColor
Голод: Дробное >=0; <=10	Hunger: Float >=0; <=10
Усталость: Дробное >=0; <=10	Tiredness: Float >=0; <=10
Счастье: Дробное >=0; <=10	Happiness: Float >=0; <=10
<i>Методы:</i>	<i>Methods:</i>
НичегоНеДелать()	DoNothing()
Спать()	Sleep()
Есть()	Eat()
Играть()	Play()
Приласкать()	DoPet()

Примечание. Это очевидные из описания атрибуты и методы. В процессе разработки мы добавим и другие.

При описании классов стараются использовать универсальный синтаксис по методологии UML, без привязки к конкретному языку программирования. Напомним, тип данных описывается в общем смысле, а не в синтаксисе языка программирования. При этом тип данных записывается полным названием после имени атрибута через двоеточие, хотя в C# типы данных обозначаются иначе.

Создание класса в C#

Создайте новое консольное приложение с именем "Fluffies".

Класс объявляется ключевым словом **class**, затем идет уникальное **имя класса** (в Pascal Style, каждое слово в имени писать с заглавной буквы), затем в фигурных скобках { } следует **описание класса**:

```
class Cat //класс Кошка
{
    //здесь будет описание класса
}
```

Однако не нужно вставлять этот код в программу вручную. Каждый класс принято описывать в отдельном cs-файле, и VisualStudio предоставляет для этого средства.

В меню "Проект" выберите "Добавить класс..." (рис. 3.3). В открывшемся окне убедитесь, что в списке выделен именно "Класс" C#, в нижней части укажите имя файла (и класса в нем) "Cat.cs".

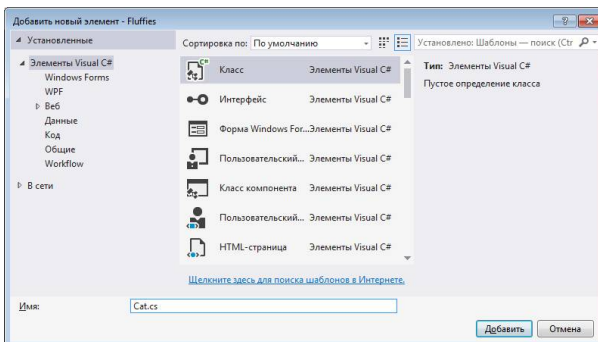


Рис. 3.3. Создание класса в VisualStudio

В результате будет создан новый файл, в котором содержится объявление класса Cat. Этот файл отобразится в обозревателе решений в составе нашего проекта Fluffies (рис. 3.4).

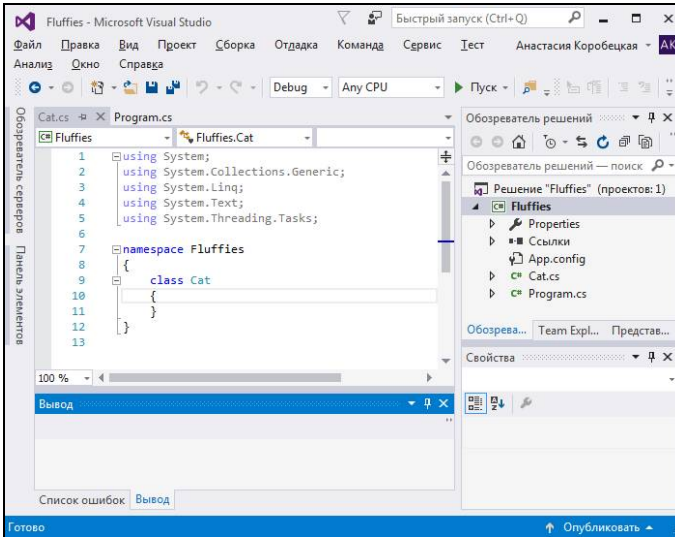


Рис. 3.4. Добавление класса Cat и файла Cat.cs в проект

Обратите внимание, что класс Cat помещен в то же **пространство имен** Fluffies. Это обеспечивает видимость класса во всей программе.

Описание класса в фигурных скобках {} должно содержать перечень всех его атрибутов и методов. Обычно атрибуты (поля) пишут в начале, а методы - после них.

Добавьте внутрь класса Cat объявление всех перечисленных в табл. 3.6 полей. **Поля** аналогичны переменным, только переменные существуют внутри методов, а поля - внутри класса и доступны всем методам этого класса.

```

namespace Fluffies
{
    class Cat
    {
        string name;
        int age;
        ConsoleColor color;
        double hunger;
        double tiredness;
        double happiness;
    }
}

```


Примечание. Имена полей принято писать с маленькой буквы, в camelStyle.

В основном файле "Program.cs" в методе Main объявим переменную myCat типа Cat. Она будет содержать **экземпляр класса**, который нужно создать командой **new**. После **new** указываем имя класса и пустые скобки.

```
1 / namespace Dummies
2 {
3     8 {
4     9     class Program
5     10 {
6     11         static void Main(string[] args)
7     12 {
8     13     Cat myCat = new Cat();
9     14 }
10 15 }
11 16 }
12 17 }
```

Cat и Cat () - не одно и то же. Cat - это класс, а Cat () - **конструктор класса**, специальный метод, который вызывается в тот момент, когда система создает экземпляр в памяти. В конструкторе можно заполнить значения по умолчанию и выполнить другую вспомогательную работу. Конструктор всегда имеет то же имя, что и класс.

Попробуем заполнить значения полей в классе Cat, например, присвоить кошке имя "Мурка":
myCat.name = "Мурка";

Напишите имя экземпляра myCat, поставьте точку, чтобы отобразить доступные поля и методы и... что-то пошло не так (рис. 3.5). В списке с выбором доступных атрибутов и методов нет полей, которые мы объявили в классе Cat. Если вы попытаетесь вручную прописать myCat.name, VisualStudio укажет на ошибку.

Дело в том, что по умолчанию значения полей доступны только *внутри класса* (внутри его фигурных скобок). Чтобы сделать поля доступными снаружи класса (из других классов и файлов), необходимо дописать перед объявлением каждого поля ключевое слово **public** (публичный, открытый, общедоступный). Подробнее о public и других уровнях доступа мы поговорим в следующей теме.

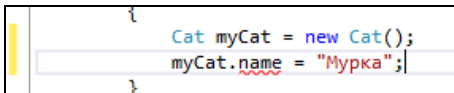
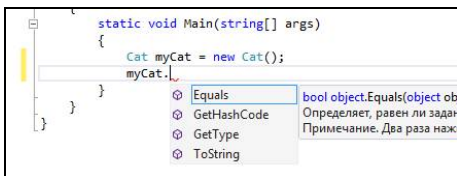


Рис. 3.5. Поле `Cat.name` недоступно в основной программе

Теперь все в порядке, можно заполнить значения полей нашей кошки из основной программы (рис. 3.6).

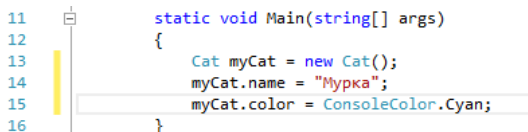
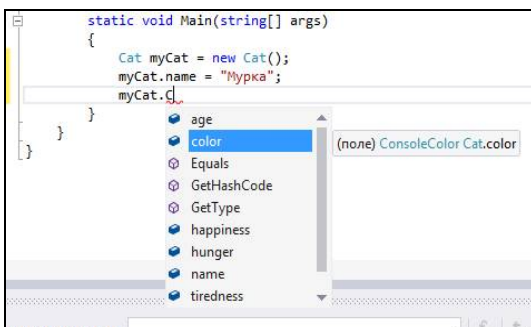
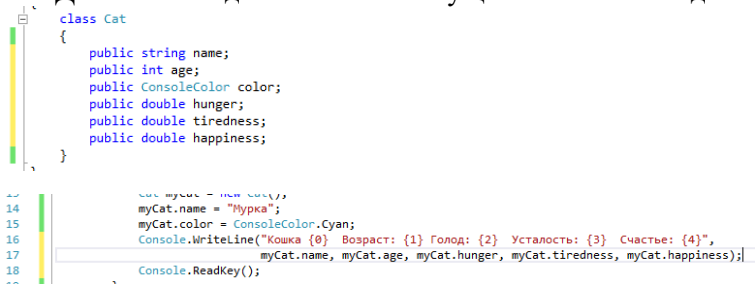


Рис. 3.6. Публичные поля доступны во внешнем файле

Давайте выведем в консоль текущее состояние созданной кошки.



Как и следовало ожидать, изначально поля равны 0 (рис. 3.7).

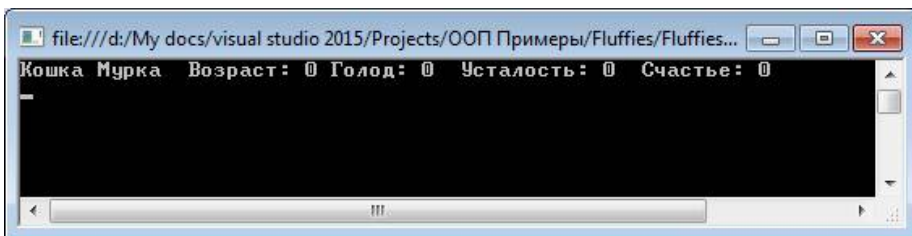


Рис. 3.7. Тестирование приложения Fluffies со значениями по умолчанию

Добавим немного визуального составляющего нашей игре. Конечно, в консоли графические возможности сильно ограничены, но с помощью спецсимволов и псевдографики можно создать достаточно узнаваемый рисунок. Например, такой:

```

| \      / |
|  \____/  |
|   V    V   |
|  (|)  (|)  |
|  >> Y <<  |
|  \____/  |
|   U    |
| %%% %%% |

```

Поскольку изображение кошки является свойством кошки, его необходимо объявить внутри класса `Cat`. Добавим метод `Show()`, который будет отвечать за вывод изображения кошки на экран. В дальнейшей разработке консольный вывод можно будет заменить на полноценную графику.

Объявим метод `Show()` с ключевым словом **public**, чтобы им можно было пользоваться в программе. Метод будет возвращать строку `string` с указанным изображением.

```

18     public string Show()
19     {
20
21     }

```

Возвращаемое значение обозначается ключевым словом **return**. При этом выполнение метода прерывается и любой код после `return` и до закрывающей метод фигурной скобки `}` будет проигнорирован.

Символ обратного слеша `\`, участвующий в изображении, является специальным (используется в комбинациях `\n`, `\t` и др.). Чтобы вывести обрат-

ный слеш в строку, его необходимо писать два раза. Из-за этого рисунок немного съехал в исходном коде, но в консоли все будет правильно.

```

18     public string Show()
19     {
20         return " |\\          /|\n" +
21                " | \\_____/ |\n" +
22                " | v    v |\n" +
23                " | (|)  (|) |\n" +
24                " ( >> Y << )\n" +
25                " \\____U____/\n" +
26                " %%%%%%%%%%\n";
27     }

```

Добавим в основную программу вывод изображения кошки заданного цвета.

```

17         myCat.name, myCat.age, m
18         Console.ForegroundColor = myCat.color;
19         Console.WriteLine(myCat.Show());|
20         Console.ReadKey();

```

Рекомендуется настроить для консоли шрифт **Lucida console** или **Consolas** (рис. 3.8).

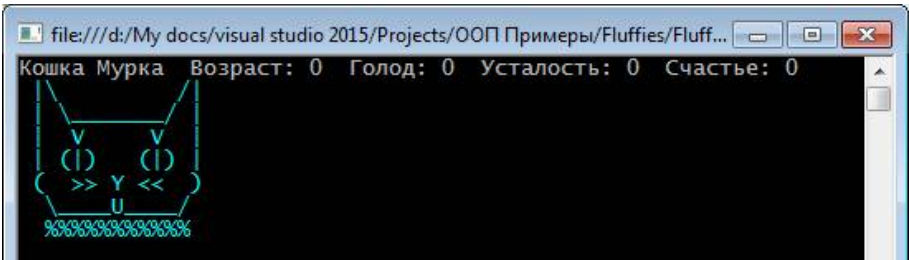


Рис. 3.8. Тестирование метода `Cat.Show()` в игре `Fluffies`

Поведение класса

Перейдем к реализации методов поведения класса, описанных в задании. Все они должны быть публичными, и все не возвращают никаких значений (тип **void** - пустой).

Начнем с метода `DoNothing()` - ничего не делать. При этом голод увеличивается на 1, а счастье снижается на 1. Объявите метод в классе `Cat` и запишите его реализацию.

```

16         public double happiness;
17
18         //ничего не делать
19         public void DoNothing()
20         {
21             hunger += 1;
22             happiness -= 1;
23         }
24
25         public string Show()

```

Добавим в основное приложение выбор этого, пока единственного, варианта действия пользователя на каждом шаге игры.

До сих пор мы использовали метод `Console.ReadKey()`, только чтобы поставить приложение на паузу. Пора применить его по назначению - чтобы узнать, какую кнопку нажал пользователь. Подцепим нажатия функциональных F1-F4 клавиш к методам класса. Пока что работать будет только F1 - ничего не делать.

Метод `Console.ReadKey()` возвращает класс `ConsoleKeyInfo`, который содержит атрибут `Key` - код нажатой клавиши. Все коды поименованы в **перечислителе** `ConsoleKey`. Выбор нажатой клавиши будем осуществлять в `switch`.

```

16         Console.ForegroundColor = myCat.Color;
19         Console.WriteLine(myCat.Show());
20         Console.WriteLine("Выберите действие:");
21         Console.WriteLine("F1 - Ничего не делать; F2 - Покормить; F3 - Играть; F4 - Приласкать");
22         ConsoleKeyInfo keyPressed = Console.ReadKey();
23         switch (keyPressed.Key)
24         {
25             case ConsoleKey.F1: myCat.DoNothing(); break;
26
27         }

```

Теперь процесс нужно зациклить. В условии сказано, что игра развивается пошагово, т.е. после каждого действия пользователя состояние кошки изменяется и обновляется информация, отображаемая на экране.

Игра продолжается бесконечно, поэтому поместим вывод информации в бесконечный цикл `while`. В начале цикла необходимо очистить экран (метод `Console.Clear()`). Еще нужно вернуть стандартный цвет консоли после того, как мы нарисовали кошку (метод `Console.ResetColor()`).

```

14         myCat.name = "Мурка";
15         myCat.color = ConsoleColor.Cyan;
16         while (true)
17         {
18             Console.Clear();
19             Console.WriteLine("Кошка {0} Возраст: {1} Голод: {2} Усталость: {3} Счастье: {4}",
20                               myCat.name, myCat.age, myCat.hunger, myCat.tiredness, myCat.happiness);
21             Console.ForegroundColor = myCat.color;
22             Console.WriteLine(myCat.Show());
23             Console.ResetColor();
24             Console.WriteLine("Выберите действие:");
25             Console.WriteLine("F1 - Ничего не делать; F2 - Покормить; F3 - Играть; F4 - Приласкать");
26             ConsoleKeyInfo keyPressed = Console.ReadKey();
27             switch (keyPressed.Key)
28             {
29                 case ConsoleKey.F1: myCat.DoNothing(); break;
30             }
31         }
32     }

```

Запустите приложение и убедитесь, что после каждого нажатия F1 голод увеличивается, а счастье снижается (рис. 3.9). При нажатии других клавиш ничего не должно измениться.

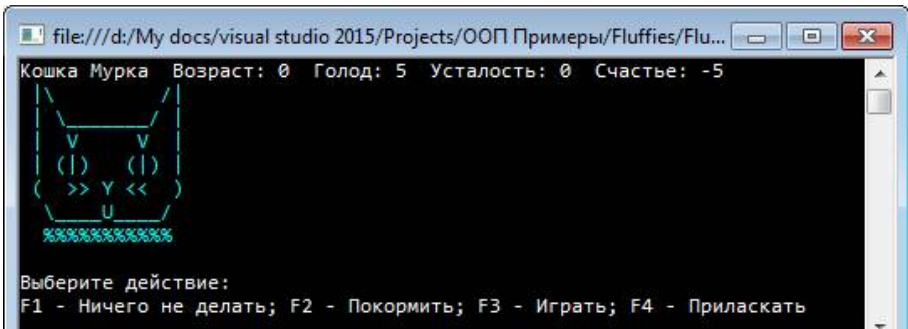


Рис. 3.9. Тестирование метода DoNothing() в игре Fluffies

Аналогично добавьте остальные методы (кормить, играть, приласкать):

```

24
25         //кормить
26         public void Eat()
27         {
28             hunger = 0;
29             tiredness += 1;
30             happiness += 1;
31         }
32

```

```

33     //играть
34     public void Play()
35     {
36         hunger += 2;
37         tiredness += 3;
38         happiness += 3;
39     }

41     //приласкать
42     public void DoPet()
43     {
44         hunger += 1;
45         tiredness += 1;
46         happiness += 1;
47     }

```

Подцепите их к клавишам F2-F4:

```

20     ConsoleKeyInfo keyInfo = Console.ReadKey();
27     switch (keyPressed.Key)
28     {
29         case ConsoleKey.F1: myCat.DoNothing(); break;
30         case ConsoleKey.F2: myCat.Eat(); break;
31         case ConsoleKey.F3: myCat.Play(); break;
32         case ConsoleKey.F4: myCat.DoPet(); break;
33     }

```

Запустите приложение и убедитесь, что все кнопки и методы работают правильно (рис. 3.10).

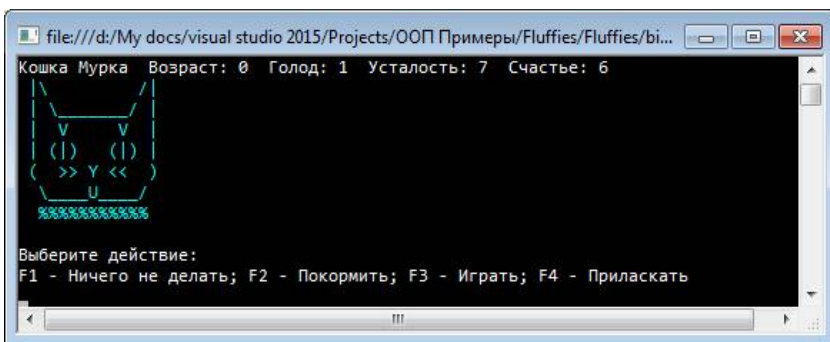


Рис. 3.10. Тестирование взаимодействия с кошкой в игре Fluffies

Тут пора вспомнить, что еще указано в описании класса: "Счастье зависит от голода и усталости. Если в конце хода голод меньше 3, то счастье увеличивается на 1. Если голод больше 5, то счастье снижается на 1, если больше 7, то на 2. Если усталость больше 5, то счастье снижается на 1, если больше 8, то на 2".

Таким образом, какое бы действие ни выбрал пользователь, все равно нужно проверить голод и усталость. Чтобы не писать один и тот же код в каждом действии, вынесем его в отдельный метод

UpdateState () - обновить состояние кошки. Этот метод будет доступен только внутри класса, поэтому писать ключевое слово public не нужно.

```
16 public double happiness;
17
18 //обновить состояние кошки
19 void UpdateState()
20 {
21
22 }
```

Добавим в метод UpdateState () проверку уровня голода:

```
18 //обновить состояние кошки
19 void UpdateState()
20 {
21     if (hunger < 3)
22     {
23         happiness += 1;
24     }
25     else if (hunger > 8)
26     {
27         happiness -= 2;
28     }
29     else if (hunger > 5)
30     {
31         happiness -= 1;
32     }
33 }
```

? Почему условия каскада if нужно записывать именно в таком порядке?

И уровня усталости:

```
29     else if (hunger > 8)
30     {
31         happiness -= 1;
32     }
33     if (tiredness > 8)
34     {
35         happiness -= 2;
36     }
37     else if (tiredness > 5)
38     {
39         happiness -= 1;
40     }
```

Еще в конце каждого хода нужно увеличить возраст кошки на 1.

```
37     else if (tiredness > 5)
38     {
39         happiness -= 1;
40     }
41     age++;
42 }
```

Будем вызывать метод UpdateState () в конце каждого действия.


```

44 //ничего не делать
45 public void DoNothing()
46 {
47     hunger += 1;
48     happiness -= 1;
49     UpdateState();
50 }
51
52 //кормить
53 public void Eat()
54 {
55     hunger = 0;
56     tiredness += 1;
57     happiness += 1;
58     UpdateState();
59 }
60
61 //играть
62 public void Play()
63 {
64     hunger += 2;
65     tiredness += 3;
66     happiness += 3;
67     UpdateState();
68 }
69
70 //приласкать
71 public void DoPet()
72 {
73     hunger += 1;
74     tiredness += 1;
75     happiness += 1;
76     UpdateState();
77 }

```

Протестируйте работу программы (рис. 3.11).

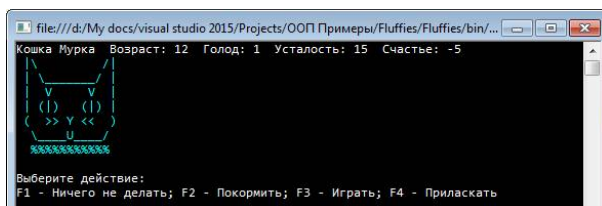


Рис. 3.11. Тестирование метода UpdateState() в игре Fluffies

Обратите внимание, нам ничего не пришлось менять в основном файле Program. Программа как бы разделилась на две части. Класс Program знает, что существует класс Cat и какие в нем есть методы и атрибуты. Но не знает, как именно эти методы работают. А класс Cat знает свою внутреннюю логику, но не знает, в какую консоль выводят его данные, какие кнопки нажимает пользователь и т.п. Это позволит после разработки консольного прототипа взять класс Cat и с мини-

маленькими изменениями подключить его к приложению с графическим интерфейсом. Или, наоборот, к консольному прототипу быстро подключить новый функционал для тестирования.

Осталось добавить метод `Sleep()` - спать. После сна усталость снижается до 0, а голод увеличивается на 4.

```
78
79
80 //спать
81 public void Sleep()
82 {
83     tiredness = 0;
84     hunger += 4;
85     UpdateState();
86 }
```

Причем выбрать это действие пользователь не может, кошка засыпает сама, когда захочет (усталость больше 8).

Это условие тоже относится к внутренней логике класса, поэтому его нецелесообразно выносить в `Program`. Добавим в класс `Cat` метод `WantToSleep()` - хочет спать, который будет возвращать тип `bool` (`true` - кошка хочет спать, `false` - не хочет).

```
86
87
88 //кошка хочет спать?
89 public bool WantToSleep()
90 {
91     return tiredness > 8;
92 }
```

Поскольку результат любого сравнения имеет тип `bool`, *не нужно* писать `if` в этом методе. Следующий код будет избыточным, так делать не следует:

```
public bool WantToSleep()
{
    // плохо - 8 строк кода вместо одной
    if (tiredness > 8)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Когда кошка спит, то пользователь не может ничего с ней делать и должен просто пропустить ход. Добавим в основную программу соответствующее условие:

```

23 Console.ResetColor();
24 // если кошка хочет спать
25 if (myCat.WantToSleep())
26 {
27     // то она спит
28     myCat.Sleep();
29     // а игрок ничего не может с ней сделать
30     Console.WriteLine("Кошка уснула. Нажмите любую клавишу");
31     Console.ReadKey();
32     continue;
33 }
34 Console.WriteLine("Выберите действие:");

```

Убедитесь, что кошка уснет, когда ее усталость достигнет нужного уровня (рис. 3.12).

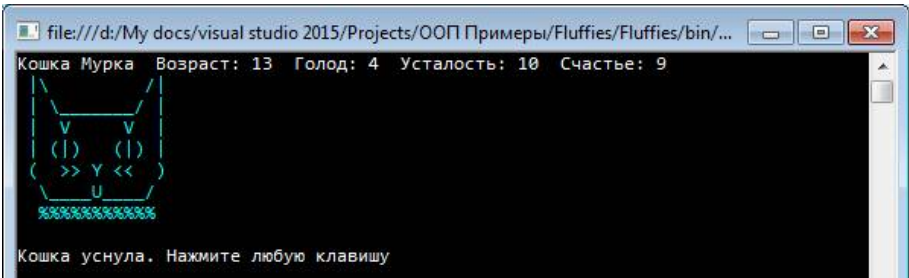
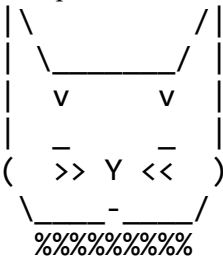


Рис. 3.12. Вывод сообщения "Кошка уснула" в игре Fluffies

Добавим еще немного красоты нашему приложению. Изменим изображение кошки, если она хочет спать.



Соответственно, в метод Show () добавляем проверку - если кошка хочет спать, то рисуем другую мордочку. При этом, поскольку мы вызываем метод WantToSleep внутри класса, то перед ним не нужно писать myCat.

```

93 public string Show()
94 {
95     if (WantToSleep())
96     {
97         return "  \\      /\n" +
98               " | \\_____/ |\n" +
99               " | v    v |\n" +
100              " | _  _ |\n" +
101              " ( >> Y << )\n" +
102              "  \\_____|_____\n" +
103              "   %%%%%%%%%%\n";
104     }
105     return "  \\      /\n" +
106           " | \\_____/ |\n" +
107           " | v    v |\n" +
108           " | ( )  ( ) |\n" +
109           " ( >> Y << )\n" +
110           "  \\____u_____\n" +
111           "   %%%%%%%%%%\n";
112 }

```

Обратите внимание, у данного if нет ветки else, хотя, на первый взгляд, она нужна. Напомним, что return прерывает выполнение метода. Поэтому если кошка хочет спать и мы попали внутрь блока if, то после return в строке 97 больше никакие действия выполнены не будут и программа не доберется до обычного изображения.

Протестируйте вывод изображения спящей кошки (рис. 3.13).

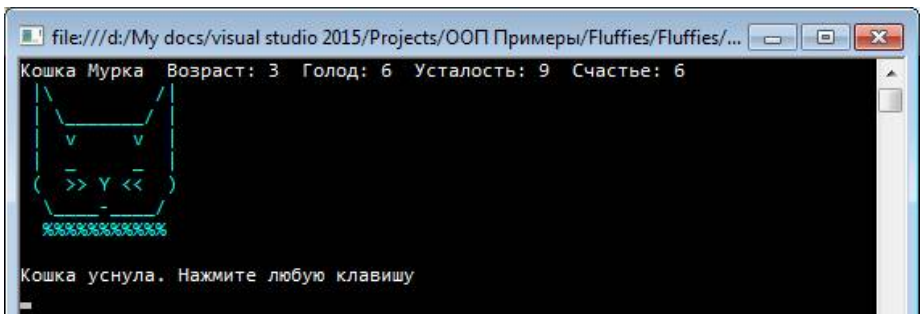


Рис. 3.13. Вывод изображения спящей кошки в игре Fluffies

Задание - добавить методы

Самостоятельно добавьте методы, проверяющие, является ли кошка голодной ($hunger > 7$) или недовольной ($happiness < 2$). Выведите соответствующие мордочки:

```

|\      /|
| \____/ |
| \    /  |
| (!)  (!) |
|  >> Y <<  |
| \____( )____/ |
| %%% %%% %%% %%% %%% %%% |

```

```

|\      /|
| \____/ |
| /    \  |
| (F)  (F) |
|  >> Y <<  |
| \____ 0 ____/ |
| %%% %%% %%% %%% %%% %%% |

```

В консоли выведите подходящий текст, например: "Мурка хочет есть" или "Мурка сердится". Вместо "Мурка" подставляйте поле `myCat.name`.

Конструктор

Напомним, что конструктор вызывается при создании экземпляра класса по команде `new`.

Пока что мы пользовались стандартным конструктором `Cat()`, чтобы создать экземпляр класса. Кроме того, мы принудительно назвали свою кошку Муркой, хотя пользователь мог дать ей другое имя.

В конструктор чаще всего передают значения полей, с которыми нужно создать экземпляр. Сделаем так, чтобы при создании кошки мы сразу спрашивали у пользователя и записывали в класс ее имя.

Переключитесь в класс `Cat`.

Добавьте объявление публичного метода с тем же именем `Cat`, без возвращаемого значения это и будет конструктор.

```

16         public double happiness;
17
18         //конструктор
19         public Cat()
20         {
21
22         }
23

```

Добавьте в круглые скобки параметр `string name` - имя кошки, которое мы сообщаем при создании.

```

16         public double happiness;
17
18         //конструктор
19         public Cat(string name)
20         {
21
22         }
--

```

В теле конструктора запишем `name` из параметра в поле `name`. Но как их различить, если и то, и другое называется `name`?

```

18         //конструктор
19         public Cat(string name)
20         {
21             // где здесь поле, а где параметр?
22             name = name;
23         }
24

```

Ответ - везде параметр, он "ближе" к коду и перекрывает собой поле с таким же именем.

Чтобы обратиться к полю (и для других целей), используется ключевое слово `this` - этот. `this` позволяет из методов класса обращаться к текущему экземпляру и его полям. То есть запись `this.name` - это обращение к кошке, которая у нас лежит в переменной `myCat`, и ее полю `name`.

```

18 //конструктор
19 public Cat(string name)
20 {
21     // слева поле, справа параметр
22     this.name = name;
23 }
24

```

Примечание. В принципе можно было просто дать параметру другое имя. Но такая запись, как получилась в примере, является общепринятой.

Теперь VisualStudio указывает на ошибку в программе, когда мы пытаемся создать экземпляр `Cat ()` без параметров (рис. 3.14) - он ожидает получить имя кошки в скобках.

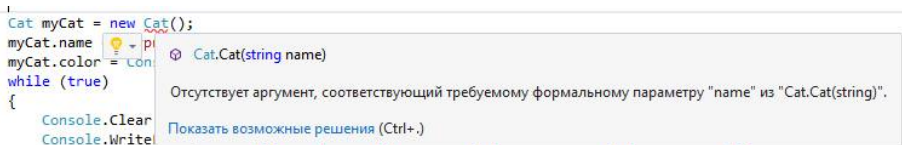


Рис. 3.14. Ошибка при вызове конструктора без параметров

Запросим имя кошки у пользователя в начале программы и передадим в конструктор.

```

13 Console.WriteLine("Придумайте имя кошки: ");
14 Cat myCat = new Cat(Console.ReadLine());
15 myCat.color = ConsoleColor.Cyan;
16

```

Выбор цвета

Самостоятельно напишите код для выбора цвета кошки. Добавьте в конструктор еще один параметр `color` и запросите у пользователя выбор одного из стандартных цветов консоли.

Пример 2. Кассовый чек v.0.2.1

В предыдущих разделах в заданиях для самостоятельной работы требовалось реализовать консольное приложение для внесения товаров в кассовый чек.

В данном примере реализуем это приложение, как и положено, через классы, их атрибуты и методы. При этом необходимо сохранить функционал приложения.

Проектирование классов "Чек" и "Товар"

В данном приложении будет два класса - чек целиком и товар (одна строка в чеке).

Атрибуты чека, строго говоря, определяются российским законодательством - это вся информация, которая должна быть на чеке. Для простоты мы возьмем только основные атрибуты: номер чека, дату и время его создания, ФИО кассира, список товаров с указанием наименования, цены и количества, сумму, оплаченную покупателем.

Обратите внимание, что вычисляемые значения - общую сумму чека и стоимость каждого товара - мы в атрибуты не включили.

В данном случае у нас нет четкого описания методов поведения. Может возникнуть вопрос - какое же поведение может вообще быть у чека, он ведь не живой и ничего делать не может. Методами в данном случае как раз будут вычисления, которые мы выполняем, а также вывод чека на печать. То есть не сам объект выполняет действие, а мы выполняем действие над ним.

Общепринятым является изображение классов на схеме в нотации UML, которая на сегодняшний день является международным стандартом. В UML класс изображается, как показано на рис. 3.15.



Рис. 3.15. Изображение класса в соответствии со стандартом UML

На начальном этапе может быть трудно сразу составить полный список всех атрибутов и методов - их можно будет добавить позже. Главное подготовить первоначальный план, от которого можно отталкиваться.

Чек будет содержать номер, дату и время выдачи, ФИО кассира, сумму оплаты, список товаров. Товар содержит название, цену, количе-

ство. При этом чек содержит список товаров, т.е. ссылается на класс Товар. Детали возможных вариантов значений нужно уточнить у заказчика.

Методы чека включают вычисления, печать, добавление товара в список. Методы товара включают вычисление стоимости и печать.

На схеме классов на рис. 3.16 показаны получившиеся атрибуты и методы с указанием типов данных и ограничений на значения. Связь между классами показана стрелкой (часть-целое).

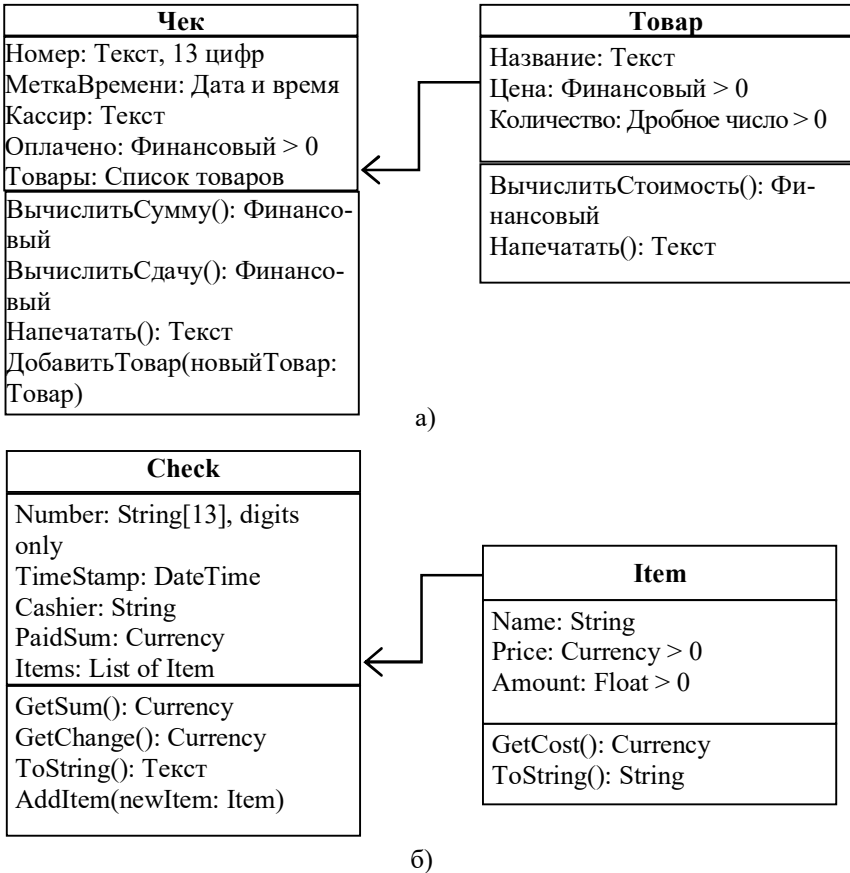


Рис. 3.16. Схема классов приложения "Кассовый чек" на русском (а) и английском (б) языках

? Почему номер чека - это текст, а не целое число?

Создание классов

Создайте в VisualStudio новый проект "CashRegister" (кассовый аппарат). Создайте файлы для двух классов: Check.cs и Item.cs.

Заполните классы основными полями, показанными на схеме классов.

Класс Item

Начнем с товара, так как это более простой класс. Добавим ему поля в соответствии с описанием. Обратите внимание на типы данных: название - это текст `string`, цена - это деньги, для них лучше всего подходит тип `decimal`, а количество может быть и целым, и дробным, поэтому берем более общий тип `double`.

```
namespace CashRegister
{
    public class Item
    {
        string name;
        decimal price;
        double amount;
    }
}
```


Причем мы не будем ставить полям публичный доступ - это плохая практика. Поля описывают внутреннюю структуру класса и не должны быть доступны напрямую, без контроля.

Добавьте публичный метод для подсчета стоимости товара. Стоимость товара - это произведение цены на количество. Результат - денежная сумма - должен иметь тип `decimal`. Вычисление записывается в одну строчку, но нужно иметь в виду, что `amount` типа `double` не может быть автоматически преобразован в `decimal`. Необходимо поставить явное приведение типов.

```
22     public decimal GetCost()
23     {
24         return price * (decimal)amount;
25     }
```

Контроль целостности данных

Всю необходимую информацию будем вносить сразу при создании экземпляра класса. Это логично - ведь в чеке могут быть только товары, у которых указаны все три поля. "Пустышки" без цены или имени там присутствовать не могут.



```

public class Item
{
    string name;
    decimal price;
    double amount;

    public Item(string name, decimal price, double amount)
    {
        this.name = name;
        this.price = price;
        this.amount = amount;
    }
}

```

Объявите конструктор класса Item с тремя параметрами, соответствующими полям.

При проектировании класса мы указали, что цена и количество должны быть строго положительными. Самое время проконтролировать **целостность данных** (т.е. их соответствие требованиям, согласованность) и выбрать стратегию действия, если введено неверное значение.

Вы можете захотеть исправить неверное значение прямо в конструкторе. Например, при отрицательном количестве присвоить значение по умолчанию - 1, а у цены просто отбросить минус. Это самое легкое решение - пара операторов ?: и все работает:

```

// если цена < 0, то отбросить минус
this.price = (price > 0) ? price : -price;
// если количество < 0, то принять его равным 1
this.amount = (amount > 0) ? amount : 1.0;

```

Но это очень плохой подход. Не надо решать за пользователя, чего он хочет. Откуда вы знаете, что имелась в виду такая же цена без знака минус? А может, он вообще мимо всех кнопок попал и ввел не то? Или в систему загрузки цен из базы данных вкралась ошибка? Или сбоят сеть и соединение с сервером нестабильное? А мы вместо того, чтобы сообщить о проблеме, тихонько подтираем данные, и пользователь даже не знает, что что-то пошло не так.

Можно предложить внести исправление, которое кажется нам уместным, но окончательно решает пользователь.

Наиболее правильным подходом будет вообще не позволять создать товар с неверной ценой или количеством - такого просто не может быть. Нужно вывести сообщение об ошибке пользователю и попросить исправить ошибочные данные.

Статические методы класса

Здесь мы подошли к новой для нас ситуации: мы хотим реализовать внутреннюю логику класса - проверка правильности ввода цены и коли-

чества. То есть это должен быть метод класса, его характеристика, а не внешний код. Основная программа должна по минимуму влезать во внутренний смысл данных класса. Ей даже не нужно знать, какие там есть поля, и тем более, какие в этих полях должны быть значения.

Примечание. Сейчас мы проверяем только на отрицательные значения, но в будущем можно добавить и другие условия. Например, ввести атрибут единицы измерения и контролировать, какие товары могут продаваться в дробных количествах, а какие не могут.

Но этот метод нужно вызвать до конструктора, т.е. до того момента, когда у нас на руках будет экземпляр класса.

Иначе говоря, мы хотим метод, привязанный к самому классу `Item`, а не к какому-то конкретному экземпляру класса. Такие методы называются **статическими** и объявляются с ключевым словом **static**.

Вы уже неоднократно сталкивались с ними, например:

- метод `Main` в вашей программе - ведь с него начинается запуск, до того, как был создан экземпляр класса программы;
- `int.Parse` и `int.TryParse`: тип `int` - это класс, который умеет преобразовывать текст в число, сами значения этого делать не умеют;
- `Array.Sort(arr)` - массив `arr` не может изменять сам себя, все операции с ним проделывает класс `Array`.

Создадим в классе `Item` два публичных статических метода `IsPriceCorrect` и `IsAmountCorrect`. Они будут возвращать `true`, если значение корректное, и `false` в противном случае.

```
15 public static bool IsPriceCorrect(decimal price)
16 {
17     return true;
18 }
19
20 public static bool IsAmountCorrect(double amount)
21 {
22     return true;
23 }
```

Примечание. Глагол `Is` в имени метода подразумевает, что он возвращает тип `bool`.

Значение на проверку нужно передать в качестве параметра. Статические классы не имеют доступа к полям - ведь они не привязаны к экземпляру класса, а только у экземпляра есть значения полей.

Мы сразу заполнили возвращаемое значение по умолчанию - `true`. То есть мы предполагаем, что "нормальным" поведением этого метода является вернуть `true`. И только если что-то пошло не так, он вернет `false`.

Добавим проверку цены:

```

15     public static bool IsPriceCorrect(decimal price)
16     {
17         if (price <= 0)
18         {
19             return false;
20         }
21         return true;
22     }

```

Можно было бы записать и в одну строчку, ведь условие (`price <= 0`) само имеет тип `bool`, никакой `if` не нужен.

```
return (price <= 0);
```

В большинстве случаев предпочтительной была бы короткая запись без `if`. Но мы хотим подчеркнуть, что нормальное поведение метода - возвращать `true`. Кроме того, если в будущем добавятся другие проверки (например, не может быть цены меньше 1 копейки), то каждому условию можно выделить свой `if` и не смешивать их в одну длинную формулу.

Аналогично проверяем количество:

```

24     public static bool IsAmountCorrect(double amount)
25     {
26         if (amount <= 0)
27         {
28             return false;
29         }
30         return true;
31     }

```

Пора протестировать работу класса `Item` - мы практически закончили его реализацию, а еще ни разу ничего не тестировали и не отлаживали.

Переключитесь в `Program.cs` и добавьте в метод `Main` создание экземпляра класса `Item`. Для экономии времени мы не будем запрашивать ввод данных у пользователя (закомментировано), а явно присвоим все значения в программе. Обратите внимание, для значения типа `decimal` нужно поставить букву `m` в конце числа, чтобы компилятор смог отличить его от `double`.

Протестируйте работу программы (рис. 3.17).

```

11     static void Main(string[] args)
12     {
13         string n = "Пакет"; //Console.ReadLine();
14         decimal p = -1.5m; //decimal.Parse(Console.ReadLine());
15         double a = 0.0; //double.Parse(Console.ReadLine());
16         if (Item.IsPriceCorrect(p) && Item.IsAmountCorrect(a))
17         {
18             Item itm = new Item(n, p, a);
19         }
20         else
21         {
22             Console.WriteLine("Ошибка! Цена и количество товара должны быть больше 0.");
23     }

```

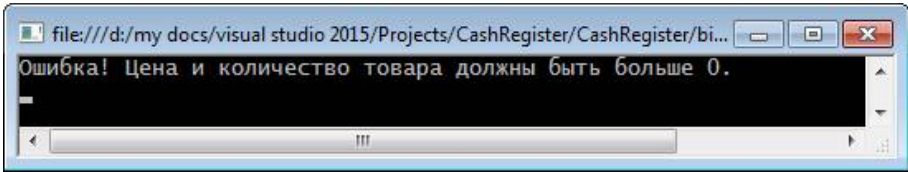


Рис. 3.17. Сообщение об ошибке при вводе неверных данных

При реализации пользовательского интерфейса следует заикнуть ввод и запрашивать данные повторно, если они неверны. При этом следует проверить цену и количество отдельно и указать, какое именно значение ошибочное.

Примечание. Контроль целостности данных при создании экземпляра можно обеспечить и другими способами. Иногда создают целый дополнительный класс-обертку, который отвечает за ввод исходных данных, их контроль и создание экземпляра основного класса. Это уместно для сложных проверок, особенно с запросами данных из БД и сети. Но для нашего примера явно слишком сложно.

Класс Check

Перейдем к классу Check. Сначала объявим **простые поля**: номер чека, дату и время создания, ФИО кассира, уплаченная сумма (по умолчанию 0).

```
namespace CashRegister
{
    public class Check
    {
        string number = new string('0', 13);
        DateTime timeStamp;
        string cashier;
        decimal paidSum = 0;
    }
}
```

Для заполнения поля `number` первоначальным значением мы использовали конструктор класса `string`: если передать ему символ и число, то он повторит этот символ указанное число раз, т.е. в `number` будет записано 13 нулей. Это соответствует длине номера чека, указанной в проекте класса.

Поле `timeStamp` (метка времени - момент создания чека) имеет тип `DateTime`. Это системный класс, который содержит основные методы для работы с датами и временем.

Добавим **конструктор класса**. Нужно передать номер нового чека и ФИО кассира. Дата и время создания чека берутся из системных настроек с помощью класса `DateTime`.

Здесь есть нюанс, не описанный нами в проекте класса: передавать номер чека в конструктор мы будем не как `string`, а как `int`. На практике номер чека обычно автоматически генерируется системой в виде последовательных чисел или случайно, т.е. представляет собой число, а в свойствах чека это число превращается в строку из 13 цифр. Такое преобразование можно выполнить с помощью метода `ToString`, сообщив ему формат `D13` - целое число, 13 знаков.

```
18     public Check(int number, string cashier)
19     {
20         this.number = number.ToString("D13");
21         this.cashier = cashier;
22         timeStamp = DateTime.Now;
23     }
```

Это избавит нас от необходимости проверять правильность номера чека.

Коллекции. Списки. Цикл `foreach`

Отличие списка от массива

Еще одно поле класса `Check` должно содержать список товаров в чеке. Пока что мы знаем только одну структуру, которая содержит перечень однотипных элементов, - это массив.

```
// массив товаров
Item[] items;
```

Но на практике массив плохо подходит для списка товаров. Дело в том, что у массива длина (т.е. число товаров в списке) четко задается в момент создания и не может быть изменена в дальнейшем.

```
// в массиве ровно 5 товаров
Item[] items = new Item[5];
```

Если вы хотите добавить товар в список, то нужно создать новый массив большей длины, скопировать туда все товары из старого массива

и заменить старый массив на новый. Тогда добавление товара в список выглядело бы примерно так:

```
// сначала в массиве ровно 3 товара
Item[] items = new Item[3];
// ввод товаров
biggerItems[0] = new Item("Масло", 128.54м, 1);
biggerItems[1] = new Item("Хлеб", 22.50м, 1);
biggerItems[2] = new Item("Колбаса", 163.99м,
1);
// добавляем 4-ый товар
Item[] biggerItems = new Item[4]; // новый
увеличенный массив
Array.Copy(items, biggerItems); // копируем из
старого в новый
// заменяем старый массив на новый
items = biggerItems;
biggerItems = null;
// ввод нового товара
items[3] = new Item("Сыр", 526м, 0.198);
```

Не очень удобно, правда? Но тут ничего не поделаешь - так уж устроена память компьютера. Массив - это ссылочный тип, и место под него выделяется в динамической памяти. Это значит, что сразу после массива начинаются другие данные, и если мы будем увеличивать массив "на месте", то эти данные затрут. Если мы хотим увеличить массив, нужно найти в памяти свободное место, зарезервировать его и переложить весь массив туда. Это мы и делаем через `biggerItems`.

Коллекции

В практике программирования на C# активно используются специальные классы - **коллекции**.

Существует много видов коллекций: список, стек, очередь, дерево, нетипизированный массив и др. Все они находятся в пространстве имен `System.Collections.Generic`. Поэтому, чтобы использовать коллекции, нужно подключить эти пространства в начале файла с директивой `using`. VisualStudio подключает его по умолчанию (серого цвета, так как еще не используется).


```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;

```

Список List

Мы рассмотрим только одну наиболее употребимую коллекцию - список `List`. Как и массив, `List` содержит однотипные объекты, но может изменять свою длину. Фактически в `C#` `List` и есть массив, только в него зашиты методы для автоматического добавления и удаления элементов. В других языках под списками могут понимать другие структуры данных.

Список имеет специфическое объявление, поскольку, как и массив, может состоять из элементов разных типов. Чтобы создать список типа `int`, нужно указать его в угловых скобках `<>` после слова `List`:

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

Список поддерживает следующие основные методы:

- void `Add(T item)`: добавление в список нового элемента `item` типа `T`;

- void `AddRange(ICollection collection)`: добавление в список всех элементов из коллекции (другого списка или массива);

- void `Insert(int index, T item)`: вставляет элемент `item` в списке на позицию `index`;

- bool `Remove(T item)`: удаляет элемент `item` из списка, и если удаление прошло успешно, то возвращает `true`;

- void `RemoveAt(int index)`: удаление элемента по указанному индексу `index`;

- void `Sort()` - сортировать список на месте;

- void `Reverse()` - развернуть список на месте;

- void `Clear()` - очистка списка (удаление всех элементов);

- int `IndexOf(T item)` - возвращает индекс первого вхождения элемента в списке (медленный последовательный поиск);

- int `BinarySearch(T item)` - быстрый бинарный поиск элемента в списке (список должен быть предварительно отсортирован);

-bool Contains(T item) - проверяет, содержит ли список элемент item.

Атрибуты:

-Count - число элементов в списке;

-Capacity - емкость, под сколько элементов уже зарезервировано место в памяти.

Например, список пользователей (тип string):

```
// создаем пустой список пользователей
List<string> users = new List<string>();
// добавляем в него элементы
users.Add("Петя");
users.Add("Вася");
users.Add("Маша");

// перебираем пользователей по номеру от 0 до Count
for (int i = 0; i < users.Count; i++)
{
    Console.Write(users[i] + " ");
}
```

Цикл foreach

У коллекций есть еще одна полезная особенность - поддержка перебора элементов **в цикле foreach** (для каждого). Вместо обычного for можно написать так:

```
// для каждого пользователя usr из списка users
foreach (string usr in users)
{
    Console.Write(usr+ " ");
}
```

Здесь переменная `usr` последовательно получит все значения из списка `users`. При этом она не будет знать о существовании других элементов в списке и какой у нее там номер. Если это и не важно, то лучше предпочесть цикл `foreach`.

Список товаров

Применим тип `List` к нашей задаче. Нам нужно создать в чеке список товаров.

```
9      public class Check
10     {
11         string number = new string('0', 13);
12         DateTime timeStamp;
13         string cashier;
14         decimal paidSum = 0;
15
16         List<Item> items = new List<Item> ();
```

Здесь ничего сложного.

Далее создадим **метод для добавления нового товара** в список. Для этого нужно передать новый `Item` в качестве параметра. Метод короткий, мы просто вызовем стандартный метод `List.Add`.

```
25     public void AddItem(Item newItem)
26     {
27         items.Add(newItem);
28     }
```

Добавьте **метод для суммирования** стоимостей товаров в списке. Воспользуемся циклом `foreach`, чтобы перебрать все товары.

```
30     public decimal GetSum()
31     {
32         decimal sum = 0m;
33         foreach (Item itm in items)
34         {
35             sum += itm.GetCost();
36         }
37         return sum;
38     }
```

Обратите внимание, метод `GetCost()` должен быть публичным, иначе вы не сможете воспользоваться им из другого класса.

Метод для вычисления сдачи `GetChange()` просто вызовет `GetSum()` и вычитет из уплаченной суммы.

```

40     public decimal GetChange()
41     {
42         return paidSum - GetSum();
43     }

```

Если вы внимательно следили за процессом разработки класса, то у вас должен возникнуть вопрос: а откуда возьмется уплаченная сумма? Ведь мы нигде не вносили ее в наш чек.

Добавим для этого еще один метод - **SetPaidSum()**.

```

45     public void SetPaidSum(decimal paidSum)
46     {
47         this.paidSum = paidSum;
48     }

```

? Почему мы не стали передавать уплаченную сумму сразу в конструкторе?

Форматированный вывод. Метод ToString()

Осталось добавить методы для вывода чека на печать.

В класс **Item** добавьте метод **ToString()**, который соберет одну строку из чека. **ToString()** - это системный метод, который используется для преобразования объекта в строку. В большинстве случаев он вызывается автоматически.

Именно благодаря нему можно записать:

```
"x = " + 10
```

и число 10 преобразуется в строку "10".

Полностью должно быть так:

```
"x = " + 10.ToString()
```

Но синтаксис C# позволяет и не писать **ToString()**.

Если мы создадим такой метод для своего класса **Item**, можно будет написать:

```
Item itm = new Item(...);
```

```
...
```

```
Console.WriteLine(item);
```

И компилятор поймет, что мы хотим вывести в консоль. Но чтобы перекрыть системную реализацию метода **ToString()**, его нужно объявить с ключевым словом **override**.

Таким образом, метод **ToString()** будет иметь вид:

```

45     public override string ToString()
46     {
47         return name + "\n"
48             + string.Format("{0, 10:n2} * {1, -6:g} = {2, 14:n2}",
49                             price, amount, GetCost());
50     }

```

С **составным форматом строк** вы уже знакомы, но здесь мы используем настройки форматирования чисел со следующим смыслом: *{индекс, число символов:внешний вид}*

Если длина числа меньше, чем число символов, то будет добавлено нужное количество пробелов. Внешний вид определяет тип числа (n2 - с разделителями тысяч и двумя знаками после запятой). Подробнее см. на MSDN.

В **классе Check** метод будет более сложным. Нам потребуется вспомогательная переменная *str*. Нужно собрать много строк и пройти по всем товарам в цикле *foreach*. Обратите внимание на форматирование даты и времени.

```

57     public override string ToString()
58     {
59         // шапка чека
60         string str = string.Format("          ЧЕК №{0}\n", number)
61             + timeStamp.ToString("dd.MM.yyyy hh:mm:ss\n")
62             + "Кассир: " + cashier + "\n"
63             + new string('*', 36) + "\n";
64         // список товаров
65         foreach (Item itm in items)
66         {
67             str += itm + "\n";
68         }
69         //итоги
70         str += new string('*', 36) + "\n"
71             + string.Format("ИТОГО: {0, 26:N2}\n", GetSum())
72             + string.Format("ОПЛАТА: {0, 26:N2}\n", paidSum)
73             + string.Format("СДАЧА: {0, 26:N2}\n", GetChange());
74         return str;

```

Можно обойтись и без цикла, воспользовавшись методом *string.Join* (это возможно только при наличии метода *ToString()* у элементов списка). Тогда и переменная *str* не потребуется.

Еще один способ форматированного вывода - **интерполяция**. Поставьте перед строкой символ *\$*, и можно будет писать не индексы, а имена переменных.

```

57     public override string ToString()
58     {
59         /*
60         // шапка чека
61         return String.Format($"          ЧЕК №{number}\n" +
62                             $"{timeStamp,28: dd.MM.yyyy hh:mm:ss}\n" +
63                             $"Кассир: {cashier}\n\n" +
64                             new string('*', 36) + "\n" +
65                             // список товаров
66                             String.Join("\n", items) + "\n" +
67                             //итоги
68                             new string('-', 36) + "\n" +
69                             $"Итого: ", -12){GetSum(),24:N2}\n" +
70                             $"Оплачено: ", -12){paidSum,24:N2}\n" +
71                             $"Сдача: ", -12){GetChange(),24:N2}");
72         */
73         // шапка чека

```

Внешний вид чека в этих двух вариантах отличается.

Попробуйте оба варианта и выберите тот, который кажется вам более удобным.

Документирующие комментарии

В С# поддерживаются специальные виды комментариев - документирующие. Они позволяют не просто пояснять смысл кода, а собирать пояснения в справочную систему.

Чтобы добавить документирующий комментарий, кликните правой кнопкой по имени класса и выберите пункт "Добавить комментарий".

```

/// <summary>
/// Кассовый чек. Содержит список купленных товаров.
/// </summary>
public class Check
{

```

Этот комментарий отображается во всплывающей подсказке IntelliSense (рис. 3.18).

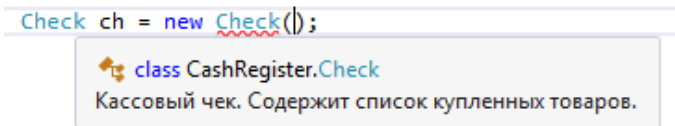


Рис. 3.18. Документирующий комментарий во всплывающей подсказке

Самостоятельно добавьте всем классам и методам документирующие комментарии, описывающие их смысл. Для методов обязательно поясните смысл передаваемых параметров (рис. 3.19).

```

28     /// <summary>
29     /// Добавляет товар в список товаров
30     /// </summary>
31     /// <param name="newItem">Товар, который будет добавлен в список</param>
32     public void AddItem(Item newItem)
33     {
34         items.Add(newItem);
35     }

```

Check myCheck = new Check(12345, "Коробецка
 myCheck.AddItem()
 void Check.AddItem(Item newItem)
 Добавляет товар в список товаров
 newItem: Товар, который будет добавлен в список

Рис. 3.19. Пояснение к параметру во всплывающей подсказке

Тестирование программы

Для тестирования мы не будем разрабатывать пользовательский интерфейс.

Не удаляйте тестовые значения, которые мы вводили для одного товара, просто прокомментируйте код.

```

13     /*string n = "Пакет"; //Console.ReadLine();
14     decimal p = -1.5m; //decimal.Parse(Console.ReadLine());
15     double a = 0.0; //double.Parse(Console.ReadLine());
16     if (Item.IsPriceCorrect(p) && Item.IsAmountCorrect(a))
17     {
18         Item itm = new Item(n, p, a);
19     }
20     else
21     {
22         Console.WriteLine("Ошибка! Цена и количество товара должны быть больше 0.");
23     }
24     */
--

```

Создайте экземпляр класса Check, заполните его товарами и выведите результат на экран. Постарайтесь перебрать разные варианты: большие и маленькие значения цены и количества, разные имена товаров.

```

26     Check myCheck = new Check(12345, "Коробецкая А.А.");
27     myCheck.AddItem(new Item("Сахар", 35.65m, 3.234));
28     myCheck.AddItem(new Item("Молоко", 59.90m, 1));
29     myCheck.AddItem(new Item("Хлеб", 25m, 2));
30     myCheck.AddItem(new Item("Минеральная вода, 0.5л", 12.23m, 12));
31     myCheck.AddItem(new Item("Таблетки от жадности", 999999.99m, 999));
32     myCheck.AddItem(new Item("Халыва", 0, 9999999));
33     myCheck.AddItem(new Item("Это я не брал", 0, 0));
34     myCheck.SetPaidSum(10000m);
35     Console.Write(myCheck);
--

```

В результате чек должен выводиться на экран в формате, показанном на рис. 3.20.

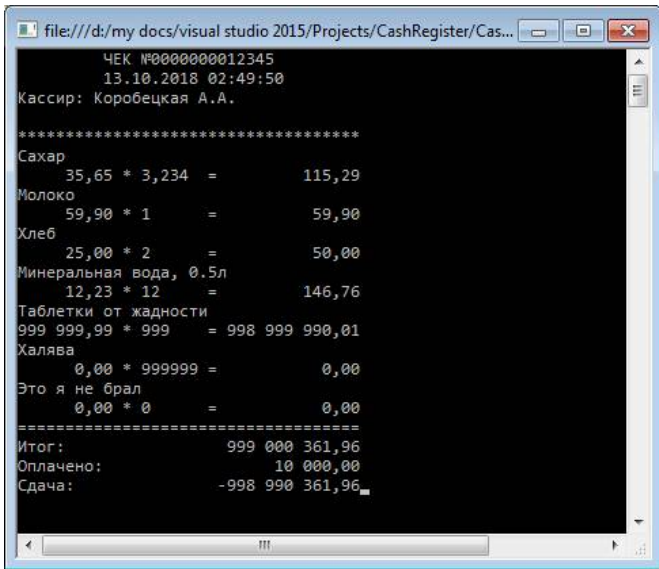


Рис. 3.20. Тестирование вывода чека на экран

Схема классов

Мы начали проектирование приложения с того, что начертили схему классов.

VisualStudio имеет встроенный формат для создания таких схем. Добавьте в проект схему классов: меню "Проект" - "Добавить новый элемент..." Найдите в списке пункт "Схема классов" (рис. 3.21).

Схема будет добавлена в ваш проект (рис. 3.22).

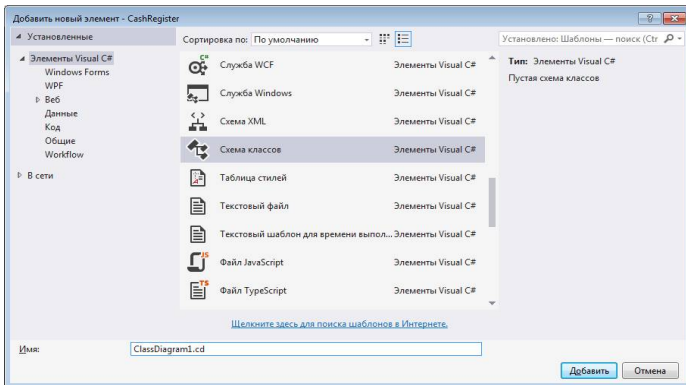


Рис. 3.21. Создание схемы классов

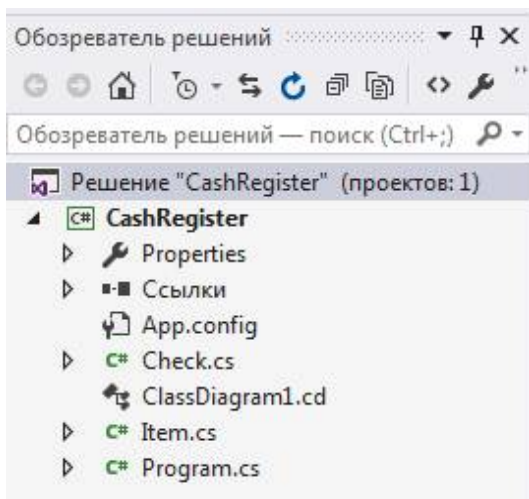


Рис. 3.22. Схема классов в "Обозревателе решений"

Откройте ее и перетащите на пустое поле классы `Check` и `Item` из "Обозревателя решений". Нажмите кнопку "Развернуть" в верхнем левом углу каждого класса, чтобы увидеть их содержимое (рис. 3.23).

Настройки полей и методов отображаются в нижней части диаграммы. Новые классы можно добавить через "Панель элементов" (свернута слева).

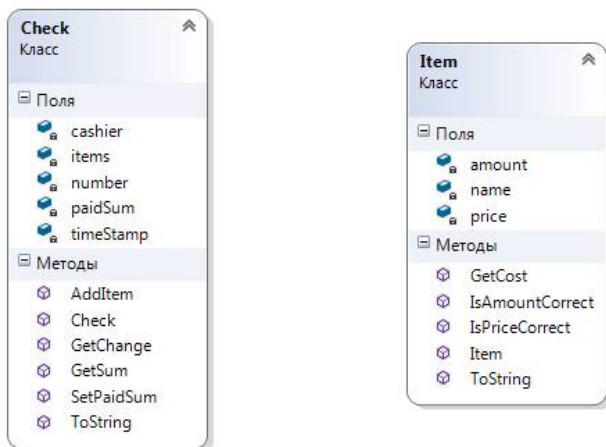


Рис. 3.23. Добавление классов на схему

Единственное, чего не хватает на схеме, это связи между чеком и товаром. Кликните правой кнопкой по полю `items` и выберите "Показывать как ассоциацию наборов" (рис. 3.24).

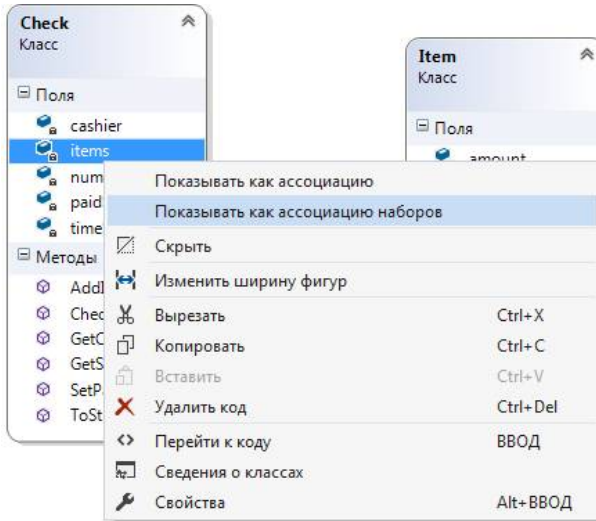


Рис. 3.24. Преобразование поля в связь-ассоциацию

В результате связь отобразится в виде двойной стрелки, как показано на рис. 3.25.

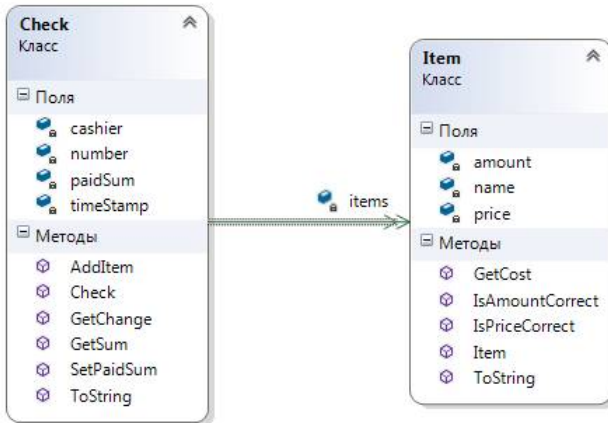


Рис. 3.25. Ассоциация наборов на схеме классов

Таким образом, структуру классов можно не писать, а создавать на визуальной схеме, а потом уже прописывать код методов в программе.

Задачи для самостоятельного решения

Задача 3.1. Геометрия

Спроектировать класс "Прямоугольник", заданный двумя парами координат: верхнего левого угла и правого нижнего угла. Реализовать методы для расчета ширины и высоты, периметра, площади, а также проверки, является ли данный прямоугольник квадратом. Отдельные методы должны перемещать прямоугольник по плоскости без изменения его размера, изменять ширину и высоту прямоугольника без изменения положения верхнего левого угла.

Реализуйте класс "Окружность", задающийся координатами центра окружности и ее радиусом. Разработайте методы вычисления площади и периметра окружности, перемещения окружности на плоскости, уменьшения площади в указанное число раз, а также метод проверки, попадает ли заданная точка внутрь окружности.

Задача 3.2. Сотрудник

Спроектируйте и реализуйте класс "Сотрудник". О сотруднике необходимо хранить ФИО, дату рождения, должность, оклад, работает ли он по совместительству. Сотруднику можно выдать зарплату (сообщить сумму к выдаче с учетом премии и НДФЛ, сумма премии вводится отдельно) и начислить страховые выплаты (30 % сверх зарплаты только для сотрудников по основному месту работы).

Создать оконный пользовательский интерфейс для заполнения полей одного экземпляра класса и вызова его методов.

Задача 3.3. Кассовый чек v.0.2.2

1. Разработайте пользовательский интерфейс для чека из примера 2. Необходимо ввести номер чека, ФИО кассира, все сведения о товарах, запросить внесенную сумму наличных. Постарайтесь сделать интерфейс максимально похожим на настоящий кассовый аппарат.

2. Доработайте метод `Check.AddItem` - если товар с таким названием уже есть в списке, то не нужно добавлять его еще раз. Вместо этого увеличьте количество этого товара. (Вспомните, когда на кассе

вам пробивают несколько товаров с одним штрих-кодом, они все собираются в одну строку).

3. Измените приложение так, чтобы в нем был не один чек, а список чеков. После завершения вывода одного чека автоматически создается новый чек со следующим по порядку номером и с тем же кассиром. Предыдущий чек удаляется с экрана. Попробуйте создать для этого новый класс - `CheckList`.

Контрольные вопросы

1. Что такое класс и экземпляр класса?
2. Что такое атрибут класса?
3. Что такое метод класса?
4. Приведите пример класса и экземпляра этого класса из жизни.
5. Что такое поле?
6. Что такое статические методы?
7. Что такое список (`List`)? Чем он отличается от массива?
8. В каких случаях применяется цикл `foreach`?
9. Что такое форматированный вывод строк? Какие в нем есть возможности?
10. Что такое документирующие комментарии?
11. Как можно задать цвет текста и фона в консоли? Какие это могут быть цвета?
12. Как создать схему классов в `VisualStudio`?

4. ПРИНЦИПЫ ООП

Объектно-ориентированное программирование строится на трех базовых принципах, единых для всех языков:

- инкапсуляция;
- наследование;
- полиморфизм.

Инкапсуляция (лат. *in capsula* - в оболочке) - сокрытие деталей реализации класса от пользователя. Иными словами, вам не нужно знать, как именно устроен класс, чтобы им пользоваться. Детали реализации от вас скрыты, а открыта только "вершина айсберга".

Данный принцип чрезвычайно распространен в повседневной жизни. Знаете ли вы во всех деталях, как устроен ваш телефон? Нет, но это не мешает вам пользоваться им. Нужно ли разбираться в работе двигателя и знать химическую формулу сгорания бензина, чтобы водить автомобиль? Нет, достаточно знать то, что нужно для управления. Но если автомобиль сломался, кому-то придется залезть под капот. Даже такой простой предмет, как табурет: чтобы сесть, вам не надо знать, из какого материала он сделан, сколько в нем гвоздей и какого диаметра ножки.

В C# мы с самого первого примера используем класс `Console`, хотя не знаем, как именно он работает. Мы знаем только, что делают его методы.

Наследование - одни классы могут наследовать от других атрибуты и методы, изменять их и добавлять свои. Как правило, наследование идет по принципу "от общего к частному", в виде иерархии.

Например, класс "Кошка", который мы рассматривали на прошлой лекции, можно представить как часть иерархии классов "Домашние животные" (рис. 4.1).

У самого верхнего (самого абстрактного) класса "Домашнее животное" есть атрибуты "Имя", "Возраст", "Пол". Эти поля будут у всех наследников данного класса, а значит, их не придется туда вписывать повторно. И у всех его наследников будут методы "Есть" и "Спать". А вот *реализация* этих методов может быть разной: корова не будет есть рыбу, как кошка, а кошка не будет клевать зерна, как канарейка. И кошка, и корова могут давать молоко, поскольку являются млекопитающими, но только корову можно подоить.

При этом конкретную Мурку - объект класса "Кошка" - можно рассматривать и как "Млекопитающее", и как "Домашнее животное".

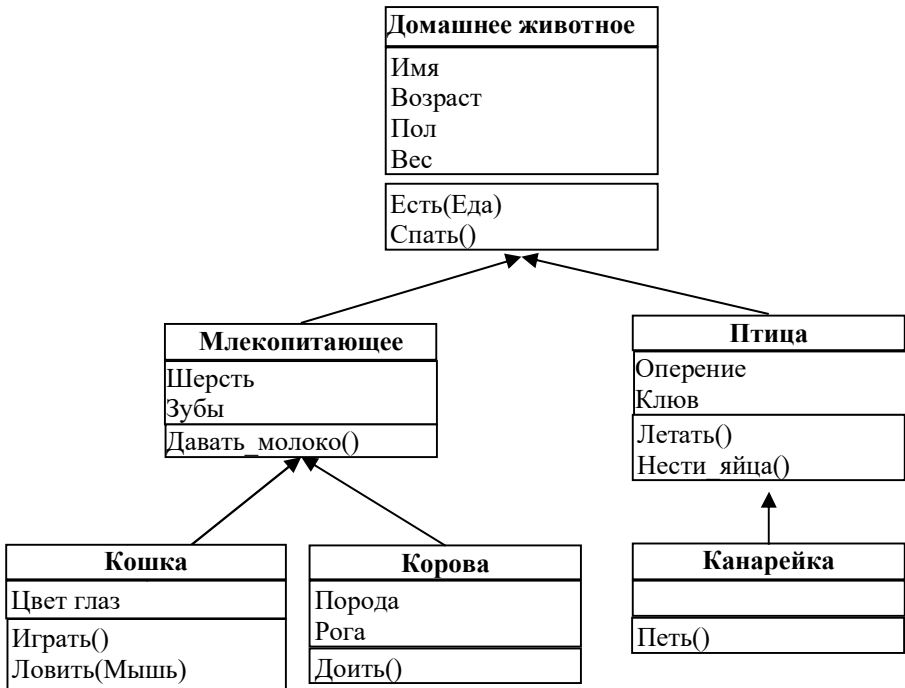


Рис. 4.1. Иерархия классов "Домашние животные"

Как именно построить эту иерархию, что выносить на верхний уровень, а что разместить на нижнем - вопрос нетривиальный, и ответ на него сильно зависит от задачи. В нашем примере мы задали птицам метод "Нести яйца". Но можно было задать "Домашнему животному" общий метод "Приносить потомство". А вместо метода "Петь" у "Канарейки" может быть метод "Издавать звуки" у "Домашнего животного", в реализации которого на нижних уровнях канарейка будет петь, кошка - мяукать, а корова - мычать.

Правильным будет то решение, которое позволит решить поставленную задачу с наименьшими усилиями как со стороны программиста, так и со стороны пользователя.

Иерархия может строиться и по другому принципу - от простого к сложному. Например:

- Табурет;
- Стул = Табурет + спинка;
- Кресло = Стул + подлокотники.

В отличие от примера с животными, кресло не является частным случаем табурета, но иерархия классов все равно работает.

В иерархии могут присутствовать **абстрактные классы**. Экземпляры таких классов не могут существовать, они слишком неопределенные. Как правило, абстрактность класса определяется наличием **абстрактных методов**.

Например, в нашей иерархии у "Домашнего животного" присутствует метод "Есть(Еда)". Мы можем описать, как и что ест кошка, корова, канарейка или курица, но как ест "просто" домашнее животное, а не какое-то конкретное, непонятно. Значит, это абстрактный метод и абстрактный класс.

С другой стороны, можно сказать, что любое животное, когда ест, поглощает пищу и прибавляет в весе. Тогда реализацией метода будет прибавить животному вес съеденной пищи, и он не будет абстрактным.

Другой пример - класс "Геометрическая фигура" с методом вычисления площади. Как посчитать площадь прямоугольника, круга или треугольника, понятно, а как вычислить площадь "просто фигуры"? Раз нет никакой конкретной реализации метода, значит, этот метод и класс в целом абстрактный. Если только мы не придумаем какую-то универсальную формулу вычисления площади любой фигуры.

Полиморфизм (гр. *многообразие форм*) - более многозначный термин, разные авторы дают ему разное определение, и у него есть несколько вариантов реализации на уровне языка программирования.

В наиболее общей формулировке методы разных классов или разные методы одного класса могут называться одинаково и решать одни и те же задачи, т.е. **реализовывать сходное поведение**. При этом пользователь не должен даже замечать разницы, как будто это одно и то же.

Пример: если вы можете сидеть, то вы можете сидеть на табурете, стуле, кресле, лавке, столе, лошади, телеге, земле или на ветке. Вам не надо для этого дополнительно учиться или менять форму тела. Достаточно знать, что то, что перед вами, пригодно для сидения (поддерживает *интерфейс* сидения и имеет метод "Сидеть").

С другой стороны, на табуретке может сидеть любой человек, а не только кто-то один. А еще может сидеть кот или кукла. При этом будет меняться поза, в которой сидят представители разных классов, т.е. один и тот же метод "Сидеть" будет *реализован* по-разному для разных типов параметров (человек, кошка, кукла).

Таким образом, существует несколько типов полиморфизма:

– **перегрузка методов** - два или более метода в классе имеют одинаковое имя, но разные списки параметров (например, кошка может играть сама с собой или с кем-то или ловить разные объекты - рис. 4.2);

Кошка
Цвет глаз
Играть () Играть (Бантик, Человек) Ловить (Мышь) Ловить (Птица) Ловить (Хвост)

Рис. 4.2. Перегрузка методов "Играть" и "Ловить" в классе "Кошка"

– **виртуализация методов** - наследники класса могут переопределять его методы, изменяя их поведение (этот тип мы затронули еще при рассмотрении наследования - общие методы "Издавать звуки" и "Приносить потомство" могут быть переопределены у классов нижнего уровня), как, например, метод "Есть" на рис. 4.3;



Рис. 4.3. Виртуализация метода "Есть" в иерархии наследования

– **интерфейсы** - классы, не являющиеся наследниками друг друга, могут реализовывать общий интерфейс и в рамках этого интерфейса предоставлять одноименные методы, как будто это одно и то же (рис. 4.4).

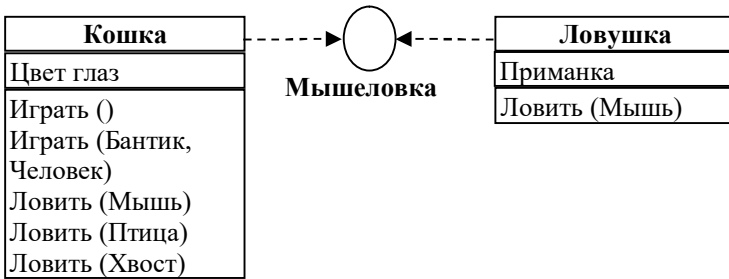


Рис. 4.4. Интерфейс мышеловки поддерживается классами, не входящими в одну иерархию наследования

Например, и массивы (класс `Array`), и списки (класс `List`) поддерживают интерфейс `IEnumerable`, а значит, для них обоих доступен цикл `foreach` и метод перебора значений.

На самом деле список вариантов полиморфизма этим не исчерпывается, иногда он даже не связан с классами. Например, оператор `+` в `C#` реализует и сложение чисел, и конкатенацию строк:

`1 + 2 = 4`
`"1"+"2"="12"`,

т.е. под одним и тем же названием скрываются разные действия.

В рамках данного пособия мы рассмотрим только первые два вида полиморфизма: перегрузку и виртуализацию методов.

Пример. Доработка прототипа игры "Fluffies"

Доработать класс "Кошка" в соответствии с принципами ООП.

Сделайте все поля приватными и добавьте свойства для доступа к ним. Необходимо проконтролировать диапазоны значений параметров.

Добавьте абстрактный класс-предок "Питомец" (`Pet`) и перенесите в него методы, общие для всех питомцев. Неопределенные методы оставьте абстрактными.

Добавьте класс "Кролик" (`Rabbit`), потомок класса `Pet`.

Кролик может выглядеть так:

```

/)__/)
(=θ.θ=)
(_____) *
/)__/)
(=-.-=)
(_____) *

```

По внешнему виду кролика трудно определить его настроение, зато он шевелит ушами (случайно) и прыгает по экрану вправо-влево.

```

(\__(\
(=θ.θ=)
(_____) *

```

Кролик не умеет играть, зато очень любит есть (табл. 4.1). Сделайте соответствующую перегрузку методов.

Таблица 4.1

Изменение показателей кролика в игре "Fluffies"

Действие \ Показатель	Ничего не делать	Спать	Есть	Ласкать
Голод	+1	+3	0	+1
Усталость	+0	0	+1	+1
Счастье	-1	+0	+2	+1

Инкапсуляция

На уровне языка программирования инкапсуляция реализуется через **области видимости** и использование **свойств**.

Области видимости

Ограничения области видимости задаются ключевыми словами, указанными в табл. 4.2.

Таблица 4.2

Ограничения области видимости

Ключевое слово	Перевод	Пояснение
private	приватная, закрытая	Члены класса доступны только внутри класса и не видны никаким другим классам
protected	защищенная	Члены класса доступны внутри класса и всем его наследникам, но не видны другим классам
public	публичная, открытая	Члены класса доступны как самому классу, так и всем другим классам

Члены класса - это поля, свойства, методы, конструкторы - все, что содержится внутри класса.

Мы уже использовали ключевое слово `public`, чтобы сделать поля и методы класса доступными для нашей программы. По умолчанию, если не указано ни одно из ключевых слов, то метод/поле/свойство считается приватным. Поэтому, если вы уберете слово `public`, например, у поля `name`, вы не сможете его использовать для вывода в консоль (по крайней мере, явно, в виде `myCat.name`).

Зачем же нужны приватные методы, если их никто не видит? Они нужны самому классу для организации его внутренней работы. В наших примерах классы довольно простые, поэтому и серьезной потребности в приватных методах нет. В реальных классах, которые порой содержат десятки полей и сотни методов, приватная область видимости позволяет облегчить программисту поиск нужных методов (легче найти один из 20, чем один из 100), а с другой стороны - запрещает лезть туда, куда не нужно.

Пока что мы просто сделали публичными все поля и методы, что является нарушением принципа инкапсуляции. Давайте разберемся, какие из них действительно должны быть открытыми, а какие следует скрыть.

Посмотрим на наш класс `Cat` через схему классов. Создайте схему классов под именем "ClassDiagramFluffies". Добавьте на схему класс `Cat`, разверните его. Выберите во всплывающем меню "Отобразить имя и тип" (рис. 4.5).

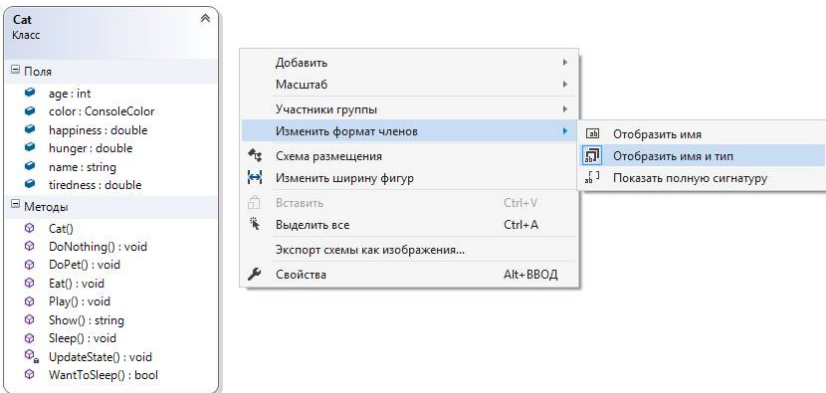


Рис. 4.5. Отображение типов данных на схеме классов

У приватных членов класса на иконке внизу отображается пиктограмма замочка. Сейчас у нас только один метод `UpdateState` является приватным - он не используется в основной программе.

Остальные методы вызываются из основной программы, поэтому должны быть публичными.

А вот **поля**, по требованиям принципа инкапсуляции, всегда должны быть приватными. Дело в том, что прямой доступ к данным не позволяет обеспечить их **целостность**. Мы никак не можем проконтролировать, что пользователь (или другой программист) запишет в публичное поле. Например, можно ввести отрицательную цену товара или поставить оценку за экзамен выше 5 и т.п.

В нашем примере значения голода, усталости и счастья должны быть от 0 до 10. Но до сих пор мы этого не обеспечили. Можно проверять диапазон значений при каждом присваивании, но это сильно удлинит и запутает код. Кроме того, всегда есть риск, что мы забудем где-то сделать проверку.

Правило! Данные должны быть защищены от прямого доступа извне.

Всегда `private`:

- все поля.

Всегда `public`:

- все конструкторы;
- метод `ToString`.

Объявите все поля приватными:

```
9      class Cat
10     {
11         private string name;
12         private int age;
13         private ConsoleColor color;
14         private double hunger;
15         private double tiredness;
16         private double happiness;
```

После этого наша программа перестанет компилироваться - ведь раньше мы использовали прямой доступ к полям, а теперь скрыли их.

Свойства

Неизбежным следствием из приватности полей следует наличие у класса свойств.

Свойство - это другой способ реализовать атрибуты класса, представляющий гибкий механизм для чтения, записи или вычисления значения.

Свойства становятся своего рода буферной зоной между пользователем и данными. Каждое свойство содержит в себе два метода (или хотя бы один из них): метод чтения данных (`get` - получить, извлечь) и

метод записи данных (set - установить, задать). В этих методах мы можем проверить, что именно пользователь пытается записать в поле или что он хочет получить, и "схватить за руку", если он делает что-то неверно.

Давайте обеспечим контроль целостности для полей hunger, tiredness и happiness.

Создайте свойство Hunger (тип данных и имя совпадают с полем hunger, только с заглавной буквы). Оно должно быть публичным и содержать внутри объявления двух методов get и set: get возвращает значение через return; set получает присваиваемое полю значение под именем value.

```
13 private ConsoleColor color;
14 private double hunger;
15
16 public double Hunger
17 {
18     get { return hunger; }
19     set { hunger = value; }
20 }
```

Добавим в set проверку: значение не должно становиться меньше 0 и больше 10.

```
16 public double Hunger
17 {
18     get { return hunger; }
19     set
20     {
21         if (value < 0)
22         {
23             hunger = 0;
24         }
25         else if (value > 10)
26         {
27             hunger = 10;
28         }
29         else
30         {
31             hunger = value;
32         }
33     }
34 }
```

Самостоятельно напишите аналогичные свойства Tiredness и Happiness.

Свойства и поля только для чтения

В некоторых случаях мы не можем изменить значения свойств - они доступны только для чтения. У таких свойств нет метода set.

В нашем примере пользователь не может принудительно изменять возраст и цвет кошки. Ведь нельзя просто взять и задать кошке, которой 3 года, возраст 5 лет или 18-летнюю кошку превратить в однолетнего котенка. То же самое с цветом - какой он есть при рождении кошки, такой и останется.

Создадим этим полям свойства, содержащие только `get`.

```
13     private int age;
14     public int Age
15     {
16         get { return age; }
17     }
18
19     private ConsoleColor color;
20     public ConsoleColor Color
21     {
22         get { return color; }
23     }
```

Но между возрастом и цветом есть принципиальная разница. Возраст автоматически увеличивается в методе `UpdateState` (внутри класса), т.е. приватное поле `age` может изменяться, просто пользователю класса не должно быть доступно присваивание.

А вот цвет - это в принципе *неизменяемое значение*. Какой цвет задали изначально (в конструкторе), такой и должен остаться.

Поэтому полю `color` нужно установить ключевое поле `readonly` - только для чтения.

```
--
19     private readonly ConsoleColor color;
20     public ConsoleColor Color
```

Поля `readonly` могут и должны присваиваться только один раз в конструкторе класса.

Автоматически реализуемые свойства

В простейшем случае, когда не требуется контролировать данные, метод `get` должен просто вернуть значение из соответствующего поля, а метод `set` - записать в него.

Например, нам не нужно проверять имя кошки - оно может быть любым.

```
11     private string name;
12     public string Name
13     {
14         get { return name; }
15         set { name = value; }
16     }
```

Возникает резонный вопрос: а зачем тогда вообще нужно свойство, если оно ничем от поля не отличается? Ответом стало появление в третьей версии языка C# автоматически реализуемых свойств. К таким свойствам не нужно создавать поля и прописывать содержимое `get` и `set`.

Удалите (совсем) поле `name`, а код свойства `Name` сократите:

```
11 public string Name { get; set; }
```

Работать будет точно так же, а кода в несколько раз меньше.

Вычисляемые свойства

Свойство не обязательно должно быть привязано к какому-то полю. Его значение может вычисляться "на лету". Начинающие программисты часто допускают ошибку, делая из таких свойств методы, возвращающие значение. Мы тоже ее допустили.

Взгляните на метод `WantToSleep()`, а также методы `IsHungry()` и `IsAngry()`, которые вы должны были реализовать самостоятельно.

Метод - это какое-то поведение, действие, совершаемое кошкой или кем-то над кошкой. А какое действие заложено в вопрос "Кошка голодная?". Это просто ее состояние, свойство.

Переделайте указанные методы в свойства только для чтения.

```
165 //кошка хочет спать?  
166 public bool WantToSleep  
167 {  
168     get { return tiredness > 8; }  
169 }  
170  
171 //кошка голодная?  
172 public bool IsHungry  
173 {  
174     get { return hunger > 8; }  
175 }  
176  
177 //кошка сердится?  
178 public bool IsAngry  
179 {  
180     get { return happiness < 3; }  
181 }
```

Правило! Если в имени метода фигурируют слова `Is`, `Get` или по смыслу он не является действием и у него нет параметров, то это должно быть вычисляемое свойство.

Обращение к свойствам

Мы переделали все поля и три метода в свойства. Сейчас наша программа не работает, так как обращаемся мы к ним по-старому.

Замените все обращения к полям на обращения к свойствам (и внутри класса, и в программе). Нужно просто заменить первую маленькую букву на заглавную. VisualStudio предложит это сделать автоматически (рис. 4.6).

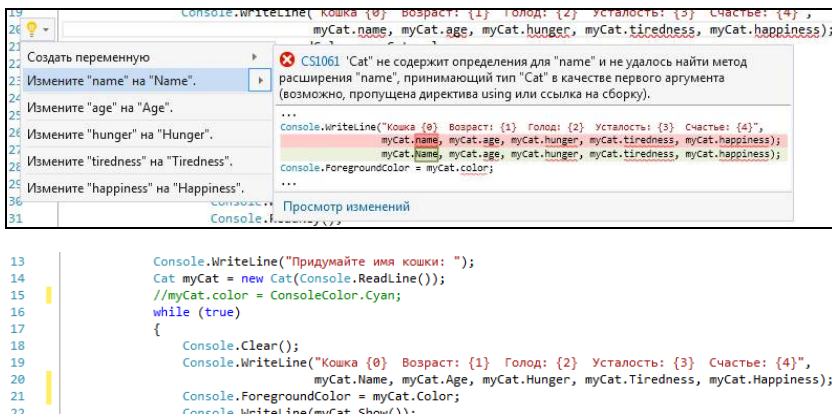


Рис. 4.6. Автоматическая замена обращения к полю на соответствующее свойство

Внутри класса VisualStudio не будет показывать ошибку, так как изнутри к приватным членам можно обращаться (для этого они и нужны). Но нам необходимо контролировать значения свойств, поэтому и внутри класса следует заменить поля на свойства. Чтобы ничего не пропустить, кликните правой кнопкой по имени поля, выберите "Переименовать" (выделяются все вхождения, как показано на рис. 4.7), исправьте имя и нажмите "Применить".

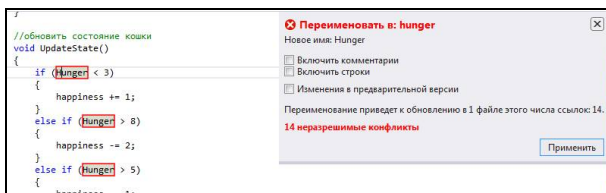


Рис. 4.7. Переименование всех вхождений имени

VisualStudio укажет на ошибку, так как мы переименовали и само поле hunger, и теперь оно совпадает со свойством. Исправьте эту

ошибку, поле должно быть с маленькой буквы. И еще - в set мы тоже присваиваем значение полю.

```
25     private double hunger;
26     public double Hunger
27     {
28         get { return hunger; }
29         set
30         {
31             if (value < 0)
32             {
33                 hunger = 0;
34             }
35             else if (value > 10)
36             {
37                 hunger = 10;
38             }
39             else
40             {
41                 hunger = value;
42             }
43         }
44     }
```

Аналогично измените обращения к остальным полям. Можно не использовать переименование, но ничего не пропустите.

Исключения - Age и Color, которые доступны только для чтения. Увеличение age++ оставьте как есть, а color нужно будет передать в конструкторе класса. Пока что просто закомментируйте присваивание в коде программы.

```
14     Cat myCat = new Cat(Console.ReadLine());
15     //myCat.color = ConsoleColor.Cyan;
16     //myCat.age++;
```

Задайте полю Color значение по умолчанию (иначе цвет будет черным и кошку не будет видно на фоне консоли).

```
19     private readonly ConsoleColor color = ConsoleColor.Cyan;
20     public ConsoleColor Color
```

Осталось исправить WantToSleep, IsHungry и IsAngry - при обращении к свойствам не нужно писать круглые скобки.

```
24     // если кошка хочет спать
25     if (myCat.WantToSleep)
26     {
27         // то она спит
28         myCat.Sleep();
```

```

184 public string Show()
185 {
186     if (IsHungry)
187     {
188         return " \\ \\ /|\n" +
189             " | \\ \\ _ /|\n" +
190             " | /   \\ \\ |\n" +
191             " | (F) (F) |\n" +
192             " | ( >> Y << )\n" +
193             " | \\ \\ _ O _/\n" +
194             " | *****\n";
195     }
196     if (IsAngry)
197     {
198         return " \\ \\ /|\n" +
199             " | \\ \\ _ /|\n" +
200             " | | \\ \\ // |\n" +
201             " | (!) (!) |\n" +
202             " | ( >> Y << )\n" +
203             " | \\ \\ _ ( ) _/\n" +
204             " | *****\n";
205     }
206     if (WantToSleep)
207     {
208         return " \\ \\ /|\n" +
209             " | \\ \\ _ /|\n" +
210             " | v   v |\n";

```

1.

Область видимости класса

Области видимости можно задавать не только для членов класса, но и для всего класса в целом.

Строго говоря, наш класс `Cat` следует объявить как `public` - ведь он должен быть доступен во всей программе.

```

7 namespace Fluffies
8 {
9     public class Cat
10    {
11        public string Name { get; set; }

```

До сих пор этого не потребовалось, поскольку мы работаем в пределах общего пространства имен `Fluffies`. Но в заданиях на самостоятельную работу вам придется писать классы в нескольких пространствах имен.

Наследование

Наследование позволяет выстроить **иерархию классов** и вынести методы, общие для нескольких классов, в единый класс-предок. Это существенно сокращает код и упрощает добавление новых объектов - зачем писать все заново, если можно только чуть доработать то, что уже есть?

В нашем примере класс-предок выделяется достаточно очевидно (рис. 4.8). Мы создали кошку для игры в уход за виртуальным питом-

цем. Значит, будут и другие виды животных (мы добавим еще кролика, но их может быть гораздо больше). Общим предком для них будет питомец (Pet).

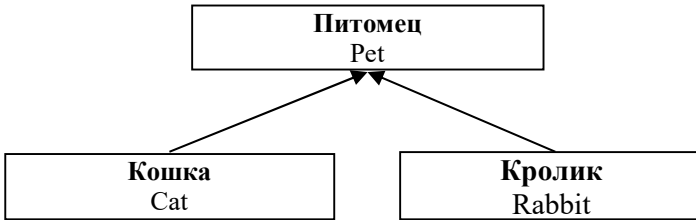


Рис. 4.8. Иерархия наследования классов в игре "Fluffies"

Такая иерархия означает, что все члены, которые у кошки и кролика одинаковые, нужно вынести в класс "Питомец", а то, что разное - поместить в классы-потомки. Например, и у кошки, и у кролика, и у любого другого питомца будет кличка и возраст - эти поля выносим в Pet. А вывод на экран (метод Show) отличается, значит, его потребуется объявить в классах отдельно.

Отметим, что классы-наследники могут дополнять или полностью изменять методы класса-предка.

Потом в иерархию можно будет добавить и другие классы (рис. 4.9).

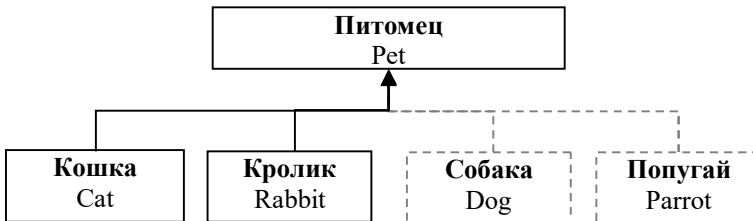


Рис. 4.9. Добавление новых классов в иерархию в игре "Fluffies"

Иерархию можно построить по-разному, в зависимости от уровней обобщения объектов. Например, можно выделить классы "Млекопитающее" и "Птица". Или "Хищник" и "Травоядное". Но это имеет смысл, только если у всех млекопитающих есть какие-то общие методы или свойства, которые отличают их от птиц.

Необходимо также знать, что **все классы** в C# входят в общую иерархию и имеют один общий корень - класс **Object**. От него наследуются абсолютно все классы. Если не указывать класс-предок при объявлении класса, то он наследуется прямо от **Object** по умолчанию. По-

этому в любом классе уже есть базовый конструктор, метод ToString (который выводит имя класса) и др. Даже такой класс `class EmptyClass {}`

на самом деле не пустой, так как содержит все методы класса Object.

Следующий вариант объявления класса полностью идентичен:

```
class EmptyClass : Object {}
```

Полностью наша иерархия классов выглядит, как показано на рис. 4.10.

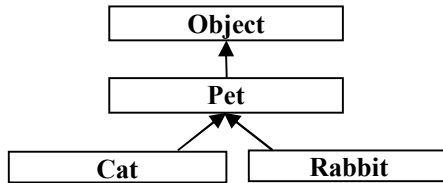


Рис. 4.10. Класс Object в иерархии классов

Но класс Object обычно не показывают на схемах, так как его атрибуты и методы очевидны для программиста. Более подробную информацию о классе Object можно найти в справке MSDN.

Класс-предок Pet

Создайте новый класс с именем Pet (как всегда, в отдельном файле).

```
7 namespace Fluffies
8 {
9     class Pet
10    {
11    }
12 }
```

Укажите, что класс Cat является его наследником. Это довольно просто - запишите Pet через двоеточие после имени класса Cat.

```
7 namespace Fluffies
8 {
9     class Cat : Pet
10    {
11        public string Name { get; set; }
12    }
13 }
```

Оба класса находятся в общем пространстве имен Fluffies, они "видят" друг друга, поэтому больше ничего дописывать не нужно.

Добавьте класс `Pet` на схему классов (рис. 4.11, для компактности поля, свойства и методы класса `Cat` свернуты). Обратите внимание, как обозначается отношение наследования: незакрашенная стрелка идет от наследника `Cat` к предку `Pet`.

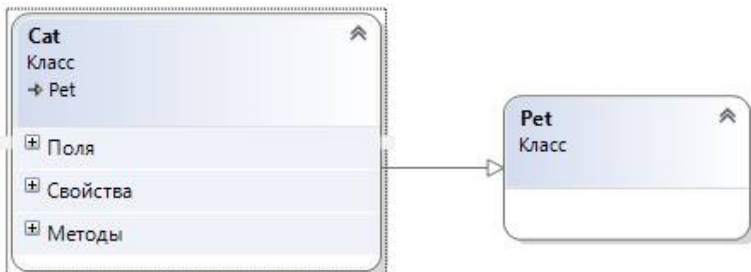


Рис. 4.11. Добавление класса `Pet` на схему классов

Давайте посмотрим, какие поля, свойства и методы нужно перенести в `Pet`. Или наоборот, какие нужно оставить в `Cat`.

У нас есть поля и свойства: имя, возраст, цвет, голод, усталость, счастье. Про все можно сказать, что эти атрибуты будут у любых питомцев. Следовательно, их место в классе `Pet`.

Перенесите поля и свойства из класса `Cat` в класс `Pet`. К сожалению, на схеме классов этого сделать нельзя, скопируйте исходный код. В результате должна получиться схема, аналогичная рис. 4.12.

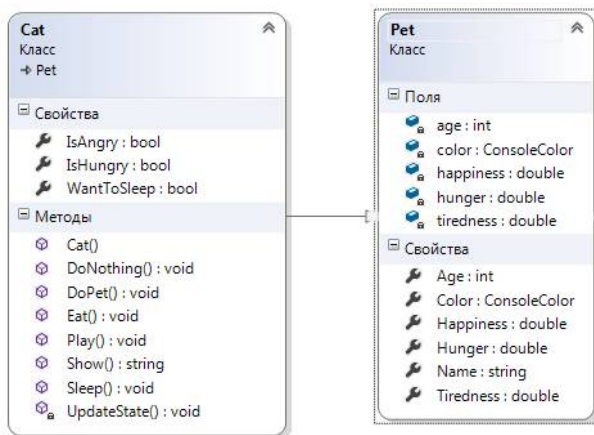


Рис. 4.12. Перенос полей и свойств в класс `Pet`

В этот момент программа перестанет запускаться, так как методы класса `Cat` потеряли доступ к полям `age` и `color` - ведь они объявлены приватными и доступны только внутри класса (рис. 4.13).

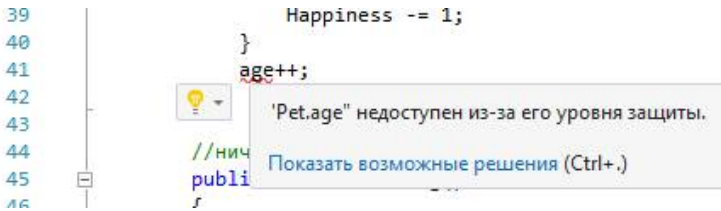


Рис. 4.13. Сообщение об ошибке из-за переноса приватного поля `age` в класс `Pet`

Но эта проблема решается сама собой - метод `UpdateState()` также будет единым для всех классов, и его нужно перенести в `Pet`.

И следом перенесем методы взаимодействия с питомцем: ничего не делать, спать, есть и ласкать. Играть умеет только кошка, этот метод переносить не нужно. Схема классов после перемещения методов показана на рис. 4.14.

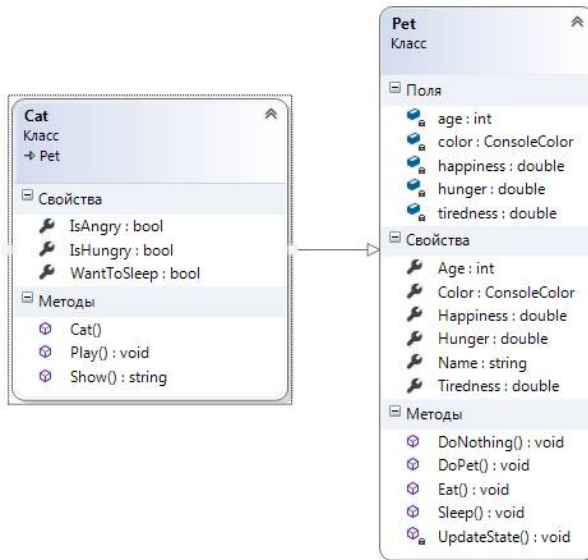


Рис. 4.14. Перемещение методов из класса `Cat` в класс `Pet`

Теперь в списке ошибок осталась только одна: приватный метод `Pet.UpdateState()` вызывается в `Cat.Play()`. VisualStudio указывает на ошибку "Метод недоступен из-за его уровня защиты" (рис. 4.15).

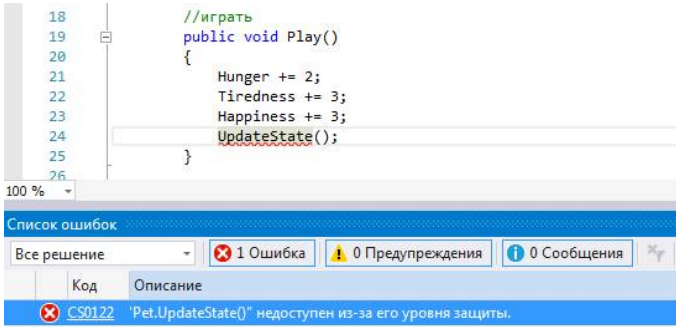
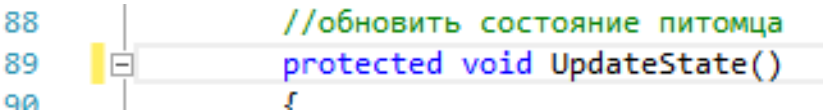


Рис. 4.15. Недоступность метода `UpdateState()` в классе `Cat`

Рассматривая инкапсуляцию, мы упомянули, но практически не обсуждали область видимости `protected` (защищенный). Она нужна как раз для таких случаев: метод должен быть виден наследникам класса, но не должен быть виден за его пределами.

Объявите метод `UpdateState` как `protected`.



После этого ошибка должна пропасть и программа запустится, как и до введения класса-наследника (рис. 4.16).

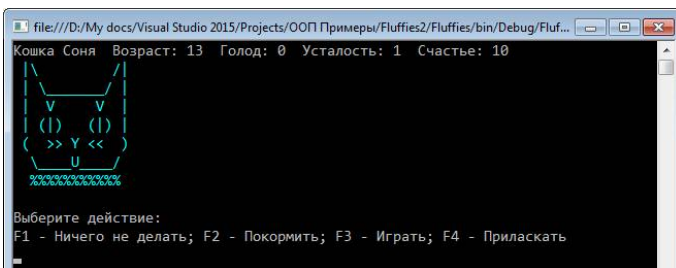


Рис. 4.16. Тестирование работы игры "Fluffies" после выделения класса `Pet`

Но на этом разработка класса-предка не заканчивается. Прежде чем мы внедрим в игру класс второго питомца - кролика, нам нужно еще кое-что доработать.

Абстрактные классы

Давайте еще раз взглянем на метод `Show()`, оставшийся в классе `Cat`. Этот метод выводит изображение кота на экран, которое естественно, отличается от кролика или собаки. Но ведь сам метод "Отобразить на экране" должен быть у любого животного. Получается, его нужно объявить в классе `Pet`.

Но как вывести на экран просто питомца - не кошку, не собаку, не птичку, а непонятно кого? Можно, конечно, извернуться и, например, сделать изображением `Pet` несколько пустых строчек или надпись "Питомец". Но правильнее будет честно признаться, что мы не знаем, как выводить на экран класс `Pet` и как реализовать соответствующий метод.

Такие методы, которые должны быть, но которые нельзя реализовать, называются **абстрактными**. У этих методов нет тела. Объявите в классе `Pet` метод `Show()` с ключевым словом `abstract` и без тела.

```
148  
149  
150  
public abstract string Show();
```

Объявление абстрактного метода - это своего рода обещание реализовать этот метод у потомков класса. Мы сообщаем, что у всех наследников `Pet` должен быть публичный метод `Show`, который возвращает текстовую строку и не требует параметров. А как именно они его реализуют, нас пока не волнует.

Наличие хотя бы одного абстрактного метода делает **абстрактным весь класс**. Его обязательно нужно тоже отметить ключевым словом `abstract` (именно из-за его отсутствия на предыдущем скриншоте отображается ошибка).

```
7 namespace Fluffies  
8 {  
9     abstract class Pet  
10    {
```


На схеме абстрактные классы показываются пунктирной границей (рис. 4.17).

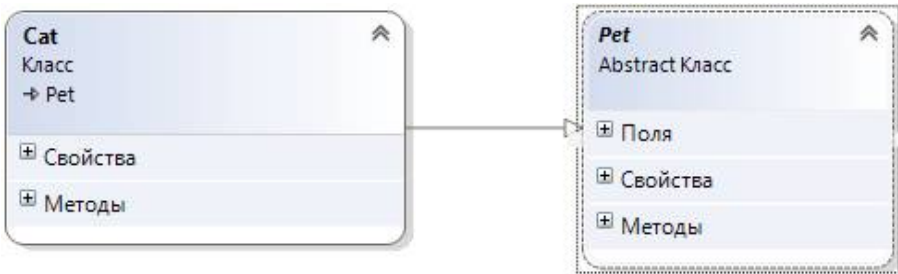


Рис. 4.17. Абстрактный класс Pet на схеме классов

Абстрактные классы нельзя создавать командой `new` - это приведет к ошибке (рис. 4.18). Вы можете объявить переменную класса Pet, но конструктор придется использовать от неабстрактного (конкретного) класса.

```
Pet myPet = new Pet();
```

Не удастся создать экземпляр абстрактного класса или интерфейса "Pet".

Рис. 4.18. Ошибка при попытке создания экземпляра абстрактного класса

Так правильно:

```
13 Console.WriteLine("Придумайте имя кошки. ");  
14 Pet myPet = new Cat(Console.ReadLine());  
15 //myCat.color = ConsoleColor.Cyan;
```

Переименуйте `myCat` в `myPet`, используя переименование по всему тексту (рис. 4.19).

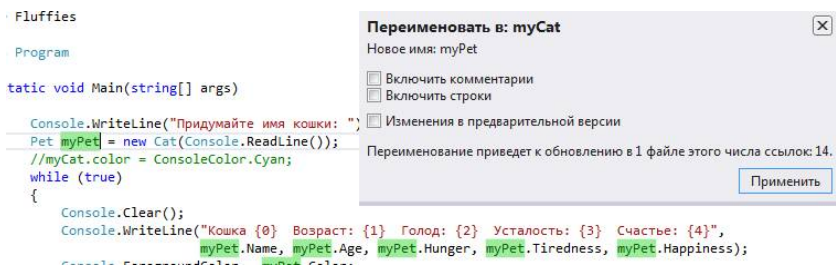


Рис. 4.19. Переименование myCat в myPet

Вывод: Объекты классов-наследников можно помещать в переменные с типом класса-предка (но не наоборот).

Операторы *as* и *is*

Если бы это было не так, для кролика и кошки пришлось бы заводить отдельные переменные, а значит - писать отдельные вызовы всех свойств и методов. И вся польза от наследования пропала бы.

Но если мы объявили переменную класса `Pet`, то и члены будут доступны только из класса `Pet`. Свойство `WantToSleep` и метод `Play()` стали недоступными, так как они объявлены в классе `Cat` (рис. 4.20).

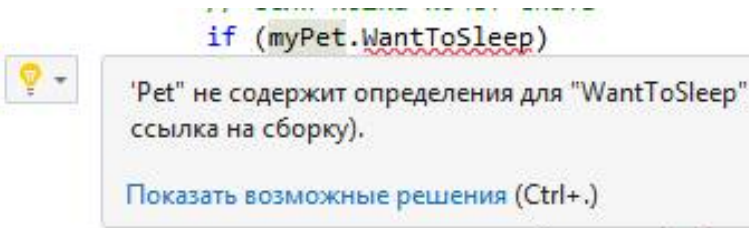


Рис. 4.20. Класс `Pet` не содержит свойства `WantToSleep`

Примечание. Мы намеренно не стали переносить их в `Pet`, чтобы продемонстрировать данную ситуацию.

Есть два основных пути решения проблемы: добавить методы в `Pet` (возможно, как абстрактные) либо как-то проверить, есть ли метод у данного наследника.

Свойство `WantToSleep` проще и логичнее перенести в класс `Pet` - ведь любое животное может захотеть спать. Скопируйте его туда (рис. 4.21).

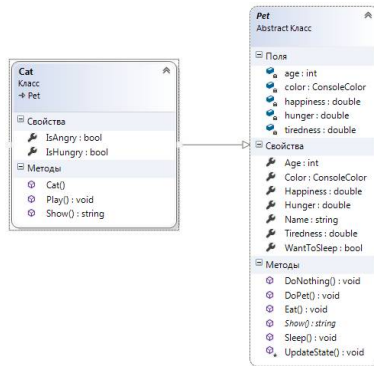


Рис. 4.21. Перенос свойства `WantToSleep` в класс `Pet`

А вот метод `Play()` должен быть только у кошки - кролик (и многие другие питомцы) не умеет играть.

Получается, что если наш питомец является кошкой, то с ним можно поиграть, а если нет, то этот метод недоступен.

Проверить класс объекта можно оператором `is`. А оператор `as` позволяет обратиться к переменной как к другому классу-наследнику.

```
41         case ConsoleKey.F3:
42             if (myPet is Cat)
43             {
44                 (myPet as Cat).Play();
45             }
46             else
47             {
48                 myPet.DoNothing();
49             }
50             break;
```

Получается, если питомец `myPet` является кошкой (`is Cat`), то с ним можно поиграть, как с кошкой (`as Cat`), а иначе он ничего не будет делать, так как не умеет играть.

Полиморфизм

Для дальнейшего улучшения наших классов потребуется рассмотреть третий принцип ООП - полиморфизм. Он может проявляться по-разному, но основная идея заключается в том, что под одним именем могут подразумеваться разные реализации.

Переопределение методов в наследовании

В нашем примере и кошку, и кролика можно отобразить на экране, но делают они это по-разному. Мы добились этого благодаря абстрактному методу, но не довели дело до конца.

Прежде всего добавим методу `Show()` в классе `Cat` ключевое слово **`override`** (перезаписать, переопределить). Этим мы явно указываем, что `Cat.Show()` должен заменить собой абстрактный `Pet.Show()`.

```
39     public override string Show()
40     {
41         if (IsHungry)
```

Обратимся к методу `Eat()`. Есть может любое животное. При этом у него голод станет равным нулю и нужно вызвать `UpdateState`. А вот изменение усталости и счастья может быть разным у разных питомцев.

Получается, что метод `Eat()` нужно разделить на две части: одна общая для всех в `Pet`:

```
123         //кормить
124         public void Eat()
125         {
126             Hunger = 0;
127             UpdateState();
128         }
```

а другая специфическая для кошки - в `Cat`:

```
33         //кормить
34         public void Eat()
35         {
36             Tiredness += 1;
37             Happiness += 1;
38         }
```

Теперь нужно связать эти две части в одно целое.

Во-первых, в классе `Pet` нужно объявить метод `Eat()` как виртуальный (`virtual`), а в классе `Cat` - как перезаписанный (`override`).

```
123         //кормить
124         public virtual void Eat()
125         {
126             Hunger = 0;
33         //кормить
34         public override void Eat()
35         {
36             Tiredness += 1;
```

Без этих ключевых слов метод `Cat.Eat()` будет считаться новым и независимым от родительского `Pet.Eat()`.

Во-вторых, в `Cat.Eat()` нужно не только изменить усталость и счастье, но выполнить все, что находится в `Pet.Eat()`. Обратиться к члену класса-предка можно с помощью ключевого слова **base**.

```

33 //кормить
34 public override void Eat()
35 {
36     Tiredness += 1;
37     Happiness += 1;
38     base.Eat();
39 }

```

base и this похожи, только base обращается к данному объекту через класс-предок, а this - через текущий класс.

По аналогии выполните переопределение метода Sleep(). Метод DoPet() объявите виртуальным, но переопределять его не нужно.

```

41 //спать
42 public override void Sleep()
43 {
44     Hunger += 4;
45     base.Sleep();
46 }

```

Перегрузка методов

Второй вид полиморфизма - **перегрузка методов**, когда в одном и том же классе встречаются два метода с одинаковым именем, но разными параметрами. Компилятор самостоятельно различает, какой метод следует применять. Никаких специальных ключевых слов использовать не требуется.

Перегружать можно только методы, так как их можно отличить по разному набору параметров. Свойства и поля в одном классе не могут быть одноименными.

Давайте добавим перегрузку метода Show. Сейчас кошка отображается в верхнем левом углу экрана. Было бы неплохо выводить ее по центру консоли или хотя бы с отступом, т.е. нужно добавить у пустых строк перед изображением и x пробелов в начале каждой строки.

Объявите в классе Pet перегрузку метода Show с двумя целочисленными аргументами x и y. Он уже не будет абстрактным - в реализации метода мы возьмем строку из Show() без аргументов, сдвинем ее и возвратим получившуюся строку.

```

153     public abstract string Show();
154
155     public string Show(int x, int y)
156     {
157         string face = Show();
158         // TO DO
159         return face;
160     }

```

Обратите внимание, мы без проблем можем вызывать абстрактный метод внутри класса - при запуске программы он уже будет переопределен.

Допишем реализацию метода. Проще всего будет разбить (`Split`) строку `face` на массив строк по символу `\n`, добавить пробелы, а потом собрать все обратно (`Join`).

```

155     public string Show(int x, int y)
156     {
157         string face = Show();
158         //разбиваем строку по символу \n
159         string[] faceSplitted = face.Split('\n');
160         for (int i = 0; i < faceSplitted.Length; i++)
161         {
162             // в начало каждой строки добавляем x пробелов
163             faceSplitted[i] = new string(' ', x) + faceSplitted[i];
164         }
165         // перед face добавляем y пустых строк и склеиваем faceSplitted
166         face = new string('\n', y) + string.Join("\n", faceSplitted);
167         return face;
168     }

```

Попробуйте вывести кошку на экран с координатами (рис. 4.22).

```

21     Console.ForegroundColor = myPet.Color;
22     Console.WriteLine(myPet.Show(20, 3));
23     Console.ResetColor();

```

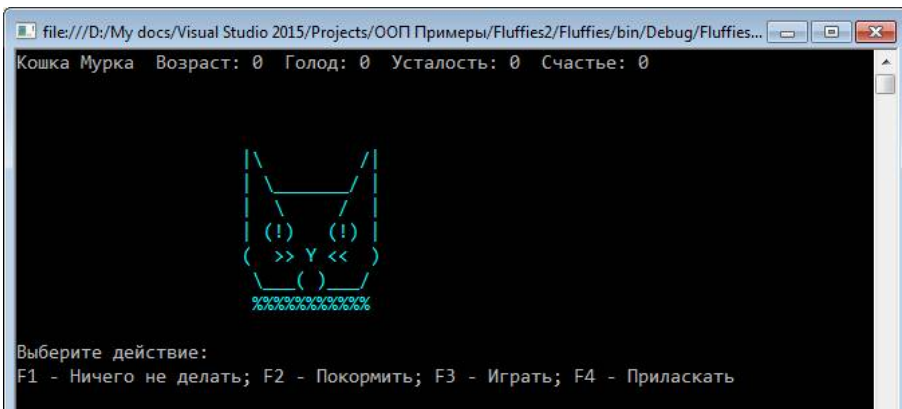


Рис. 4.22. Вывод изображения кошки в заданных координатах

Перегрузка и переопределение конструктора

Последний штрих к нашему классу - это конструктор. Во-первых, сейчас при создании кошки мы сообщаем только имя, а цвет остается по умолчанию. Было бы логично цвет тоже передавать в конструктор. Во-вторых, все питомцы будут создаваться идентично: передать в конструктор имя и цвет, записать их в переменные. Но, в отличие от прочих методов, имя конструктора всегда совпадает с именем класса. А для переопределения методы должны иметь одно имя.

Сначала добавим конструкторы в класс `Pet`. Их будет два: с одним и с двумя аргументами.

```
9      abstract class Pet
10     {
11         public Pet(string name)
12         {
13             this.Name = name;
14         }
15
16         public Pet(string name, ConsoleColor color)
17         {
18             this.Name = name;
19             this.color = color;
20     }
```

Заметьте, во втором конструкторе повторяется строка про `name` из первого. Одна строка - это немного, но реальные конструкторы могут быть довольно длинными. `Pet(name, color)` должен повторить все, что делает `Pet(name)` - давайте запишем это в явном виде.

```
11     public Pet(string name)
12     {
13         this.Name = name;
14     }
15
16     public Pet(string name, ConsoleColor color) : this(name)
17     {
18         this.color = color;
19     }
```

То есть мы можем обратиться к другой перегрузке метода через `this`.

Теперь перейдем к классу `Cat`. В нем должно быть два точно таких же конструктора, но называться они должны иначе. Это можно реализовать через `base`. Старый конструктор `Cat` нужно удалить.

```
9      class Cat : Pet
10     {
11         public Cat(string name) : base(name) { }
12         public Cat(string name, ConsoleColor color) : base(name, color) { }
13     }
```

Добавьте в основную программу выбор цвета и вызов соответствующего конструктора. Проверьте, что пользователь ввел допустимый номер цвета.

```
11     static void Main(string[] args)
12     {
13         Console.WriteLine("Придумайте имя питомца: ");
14         string name = Console.ReadLine();
15         Console.WriteLine("Выберите цвет (1-15): ");
16         int color = int.Parse(Console.ReadLine());
17         Pet myPet;
18         if (1 <= color && color <= 15)
19         {
20             myPet = new Cat(name, (ConsoleColor)color);
21         }
22         else
23         {
24             Console.WriteLine("Такого цвета не существует! Назначен цвет по умолчанию.");
25             myPet = new Cat(name);
26         }
27     }
```

Убедитесь, что на данном этапе программа работает корректно - можно выбрать цвет, выполняются все действия (рис. 4.23).

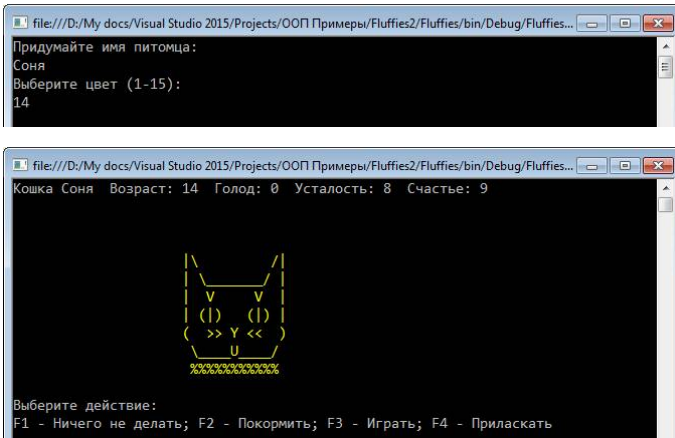


Рис. 4.23. Выбор цвета кошки в игре "Fluffies"

В итоге нам пришлось внести довольно много изменений в код - а все из-за того, что классы изначально были спроектированы без учета принципов ООП. Если вы будете следовать им сразу, то и проблем не будет.

Зато теперь мы можем с легкостью добавить новый класс - кролика.

Новый класс-наследник Rabbit

Создание класса

Давайте попробуем создать новый класс через схему классов. Откройте диаграмму и отобразите панель элементов, если она скрыта. Выберите пункт "Класс" (рис. 4.24) и разместите новый класс на схеме.

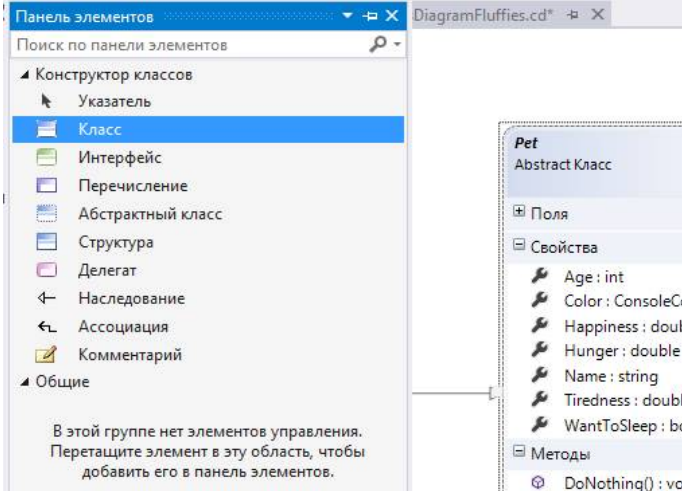


Рис. 4.24. Создание класса через схему классов

Задайте новому классу имя Rabbit, доступ - по умолчанию (т.е. не указывается), создайте новый файл и тем же именем Rabbit, как показано на рис. 4.25.

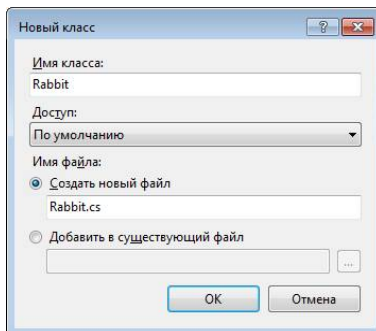


Рис. 4.25. Создание файла нового класса

Покажите, что Rabbit наследуется от Pet - проведите стрелку наследования от Rabbit к Pet (рис. 4.26).

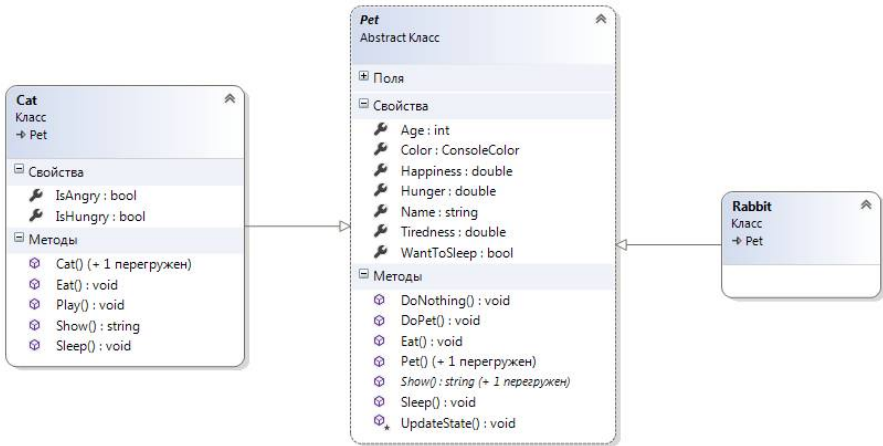


Рис. 4.26. Наследование класса Rabbit от Pet на схеме классов

Добавьте кролику два конструктора и методы Eat (), Sleep (), Show (). Настройте их с помощью панели "Сведения о классах" (рис. 4.27, 4.28).

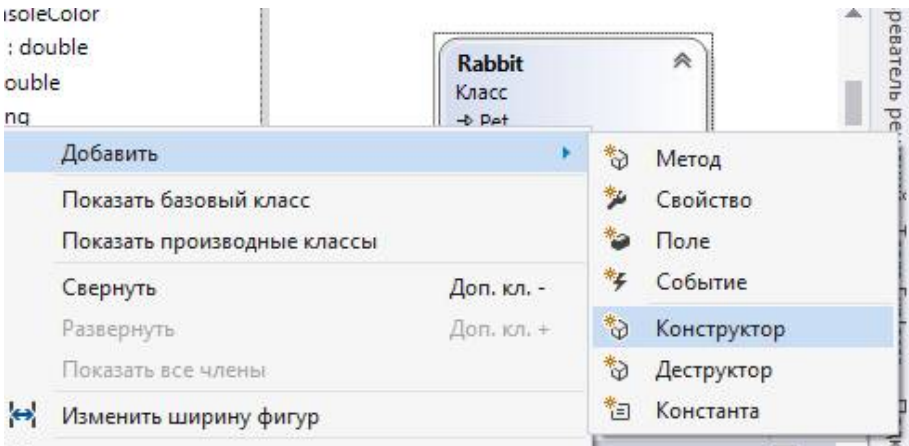


Рис. 4.27. Добавление членов класса Rabbit

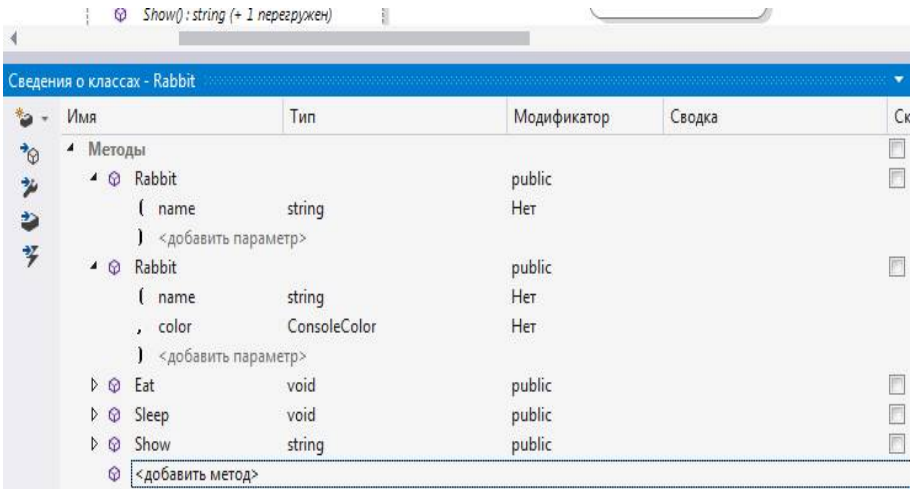


Рис. 4.28. Панель "Сведения о классах"

В результате класс Rabbit должен принять вид, аналогичный рис. 4.29.

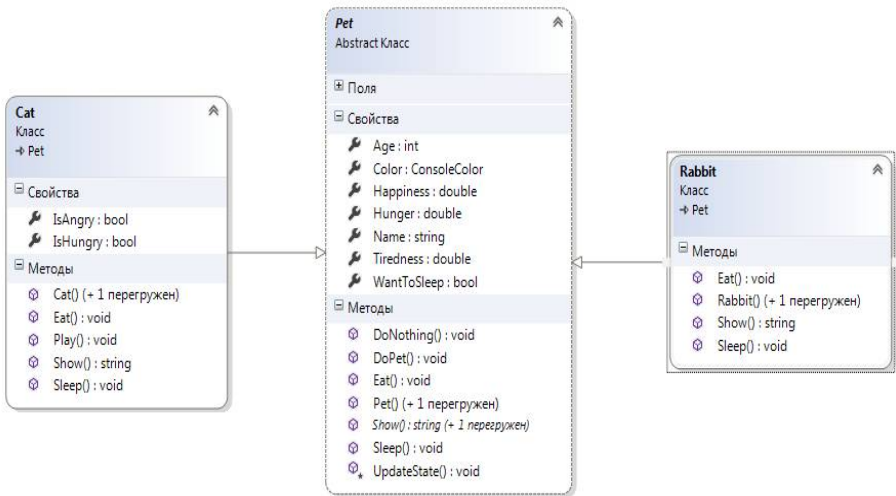


Рис. 4.29. Содержимое класса Rabbit на схеме классов

Откройте файл Rabbit.cs - в нем уже объявлены все указанные методы, но вместо тела у них стоит выброс исключения.

```

3   class Rabbit : Pet
4   {
5       public Rabbit(string name, ConsoleColor color)
6       {
7           throw new NotImplementedException();
8       }
9
10      public Rabbit(string name)
11      {
12          throw new NotImplementedException();
13      }
14
15      public void Eat()
16      {
17          throw new NotImplementedException();
18      }
19
20      public void Sleep()
21      {
22          throw new NotImplementedException();
23      }
24
25      public string Show()
26      {
27          throw new NotImplementedException();
28      }
29  }

```

Реализуйте методы согласно заданию. Не забудьте ключевое слово `override`.

```

7   {
8       class Rabbit : Pet
9       {
10          public Rabbit(string name, ConsoleColor color) : base(name, color) { }
11          public Rabbit(string name) : base(name) { }
12
13          public override void Eat()
14          {
15              Tiredness += 1;
16              Happiness += 2;
17              base.Eat();
18          }
19
20          public override void Sleep()
21          {
22              Hunger += 3;
23              base.Sleep();
24          }

```

```

26 public override string Show()
27 {
28     if (WantToSleep)
29     {
30         return " /)__)\\n" +
31             "(=. .=)\\n" +
32             "(____) *";
33     }
34     return " /)__)\\n" +
35         "(=0.0=)\\n" +
36         "(____) *";
37 }

```

Случайные числа

По заданию требуется, чтобы кролик случайно шевелил ушами, т.е. иногда уши у него будут наклонены в одну сторону, а иногда в другую.

Для генерации случайных чисел используется **класс Random**. Необходимо создать экземпляр класса и вызвать один из четырех методов:

- Next () - случайное целое число;

- Next (int maxValue) - случайное целое число в диапазоне от 0 до maxValue (не включая его);

- Next (int minValue, int maxValue) - случайное целое число в диапазоне от minValue до maxValue (включая minValue, но не включая maxValue);

- NextDouble () - случайное дробное число от 0.0 до 1.0.

Например:

```

Random r = new Random();
// с вероятностью 30%
if (r.NextDouble() < 0.3)
{
    Console.WriteLine("Выпало <0.3");
}

```

У нас оба положения ушей могут получиться с вероятностью 50 %. Отдельную переменную для Random заводить не будем.

```

26     public override string Show()
27     {
28         string ears = ((new Random()).NextDouble() < 0.5) ? " /_/" : " (\_(\\";
29         if (WantToSleep)
30         {
31             return ears + "\n" +
32                 "(=.-.)\n" +
33                 "(____)*";
34         }
35         return ears + "\n" +
36             "(=0.0=)\n" +
37             "(____)*";
38     }

```

Класс Rabbit готов! Это было несложно, не правда ли? В этом и состоит мощь ООП - легко изменять, добавлять и адаптировать программу под новые задачи. Конечно, если все сделать правильно, плохо спроектированные классы только добавляют проблем.

Выбор питомца в программе

Для начала просто попробуйте создать в программе кролика вместо кошки.

```

17     Pet myPet;
18     if (1 <= color && color <= 15)
19     {
20         myPet = new Rabbit(name, (ConsoleColor)color);
21     }
22     else
23     {
24         Console.WriteLine("Такого цвета не существует! Назначен цвет по умолчанию.");
25         myPet = new Rabbit(name);
26     }
27

```

Чтобы кролик "бегал" по экрану, добавим генерацию случайной координаты x.

```

34     Console.ForegroundColor = myPet.Color;
35     Random x = new Random();
36     Console.WriteLine(myPet.Show(x.Next(50), 3));
37     Console.ResetColor();

```

Убедитесь, что все работает (рис. 4.30 и 4.31).

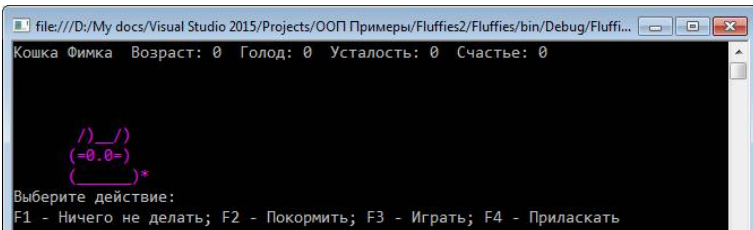


Рис. 4.30. Добавление кролика в игре "Fluffies"

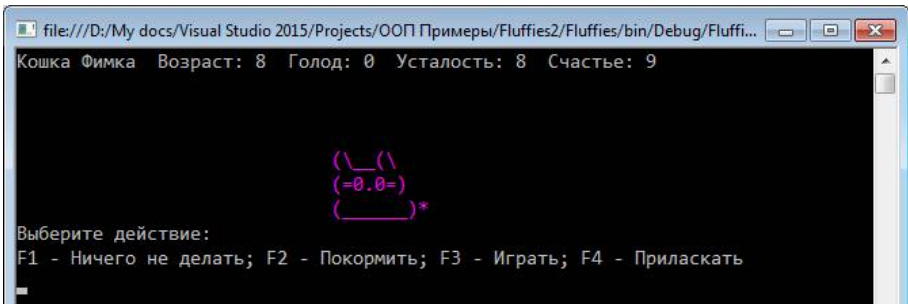


Рис. 4.31. Случайное перемещение кролика в игре "Fluffies"

Все хорошо, не считая того, что в верхней строке наш кролик называется кошкой. Конечно, мы можем просто добавить оператор `is`, чтобы проверить, какой питомец сейчас создан, но когда их станет больше двух, проверять будет проблематично.

Добавим нашим классам еще одно свойство `Specie`, которое будет возвращать строку с названием вида питомца.

В классе `Pet` это свойство будет абстрактным.

```

9      abstract class Pet
10     {
11         public abstract string Specie { get; }
12     }

```

А в классах `Cat` и `Rabbit` - возвращать соответствующее значение.

```

9      class Cat : Pet
10     {
11         public Cat(string name) : base(name) { }
12         public Cat(string name, ConsoleColor color) : base(name, color) { }
13
14         public override string Specie { get { return "Кошка"; } }
15     }

```

```

8      class Rabbit : Pet
9     {
10         public Rabbit(string name, ConsoleColor color) : base(name, color) { }
11         public Rabbit(string name) : base(name) { }
12
13         public override string Specie { get { return "Кролик"; } }
14     }

```

Везде в тексте программы замените слово "кошка" на вызов свойства `Specie`.

```

        Console.WriteLine("{5} {0} Возраст: {1} Голод: {2} Усталость: {3} Счастье: {4}",
            myPet.Name, myPet.Age, myPet.Hunger, myPet.Tiredness, myPet.Happiness,
            myPet.Specie);
        Console.ForegroundColor = myPet.Color;
    }
}

```

// а игрок ничего не может с ней сделать

```

Console.WriteLine("{0} {1} спит. Нажмите любую клавишу", myPet.Specie, myPet.Name);
Console.ReadKey();
}
}

```

Добавьте выбор вида питомца в программу.

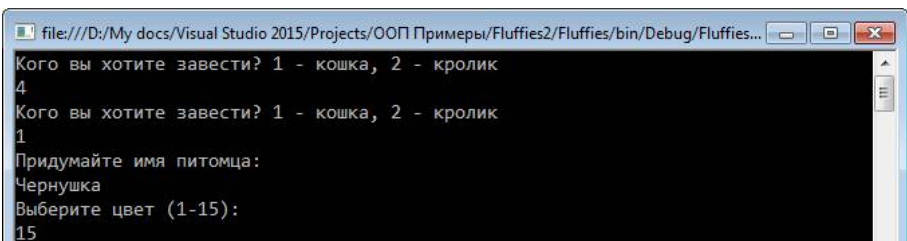
```

    static void main(string[] args)
    {
        string specie;
        do {
            Console.WriteLine("Кого вы хотите завести? 1 - кошка, 2 - кролик");
            specie = Console.ReadLine();
        } while (specie != "1" && specie != "2");

        Pet myPet;
        if (1 <= color && color <= 15)
        {
            if (specie == "1")
            {
                myPet = new Cat(name, (ConsoleColor)color);
            }
            else
            {
                myPet = new Rabbit(name, (ConsoleColor)color);
            }
        }
        else
        {
            Console.WriteLine("Такого цвета не существует! Назначен цвет по умолчанию.");
            if (specie == "1")
            {
                myPet = new Cat(name);
            }
            else
            {
                myPet = new Rabbit(name);
            }
        }
    }
}

```

Протестируйте работу программы (рис. 4.32 и 4.33).



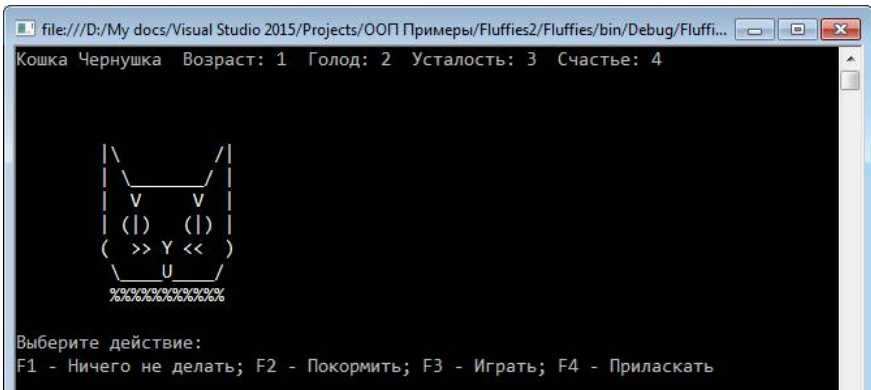


Рис. 4.32. Выбор кошки в игре "Fluffies"

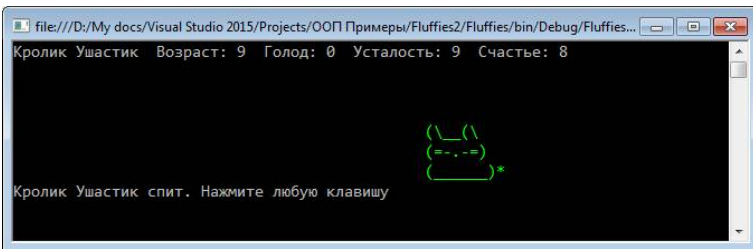
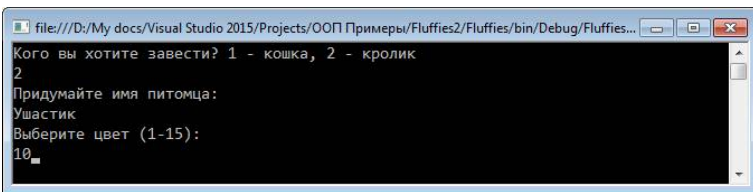


Рис. 4.33. Выбор кролика в игре "Fluffies"

На этом мы завершаем разработку игры "Fluffies."

Задачи для самостоятельного решения

Задача 4.1. Кассовый чек v.0.3.1

Доработайте приложение "Кассовый чек" из предыдущего раздела в соответствии с принципами ООП.

1. Добавьте свойства для доступа к полям классов `Check` и товар `Item`.
2. Преобразуйте в вычисляемые свойства методы, определяющие стоимость товара, сумму чека, сдачу.

3. Добавьте чеку свойство `Count`, возвращающее количество товаров в чеке.

4. Добавьте чеку метод `Item Search(...)`, который будет находить и возвращать первый товар в списке, удовлетворяющий заданному условию. У него должно быть три перегрузки:

- `Item Search(string itemName)` - поиск только по имени;

- `Item Search(string itemName, decimal itemPrice)` - по имени и цене;

- `Item Search(string itemName, decimal itemPrice, double itemAmount)` - по имени, цене и количеству.

5. Добавьте методам `AddItem(...)` и `RemoveItem(...)` несколько перегрузок:

- `AddItem(Item newItem)` и `RemoveItem(Item newItem)` - добавляет/удаляет заданный объект в списке;

- `RemoveItem(string itemName)` - должен через вызов `Search(string itemName)` найти в списке товаров `Item` с таким же именем и удалить его из чека;

- `AddItem(string itemName, decimal itemPrice, double itemAmount)` - если такой товар (с таким же именем и ценой) уже есть в чеке, то прибавить `itemAmount` к его количеству, иначе создать новый объект `Item` и добавить его в список.

6. Подключите реализованные методы к программе, чтобы перед оплатой чек можно было отредактировать. Считывайте для этого управляющие кнопки с клавиатуры: `Insert` - добавить товар, `Delete` - удалить товар, `End` - завершить редактирование чека и внести оплату.

7. Добавьте классу `Item` еще один конструктор, без указания количества (по умолчанию = 1). Если пользователь не указывает количество товара (вводит пустую строку), то вызывать этот конструктор.

Задача 4.2. Оплата труда

Создайте три класса, описывающие работников с почасовой оплатой и фиксированной оплатой.

О работнике хранится только ФИО и ставка оплаты труда. У работников с почасовой оплатой указывается оплата в час, у работников с фиксированной оплатой - оклад в месяц.

У всех работников есть метод для расчета оплаты труда за месяц. У "повременщиков" зарплата = 20 дней * 8 часов * оплата в час; у работников с фиксированной оплатой зарплата = оклад. У всех сотрудников из зарплаты вычитается НДФЛ = 13 %. Оставшуюся сумму работник получает на руки.

Создайте в программе список из 6-7 сотрудников с разными типами. Выведите на экран таблицу с зарплатной ведомостью этих сотрудников со столбцами №, ФИО, вид оплаты, ставка, зарплата за месяц, НДФЛ, оплата на руки.

Используйте символы псевдографики (`— † =` и др.) для рисования линий. Выравнивайте числа и текст с помощью составного формата строк.

(*) Подведите итоги по таблице. Сделайте адаптивную ширину столбца ФИО (ширина = самая длинная ФИО).

№	ФИО	Оплата	Ставка	Зарплата	НДФЛ	Получено
1	Александров З.И.	фикс.	27 134,00	27 134,00	3 527,42	23 606,58
2	Кунатенко А.В.	почас.	172,00	27 520,00	3 577,60	23 942,40
3	Петров Н.А.	почас.	302,00	48 320,00	6 281,60	42 038,40
4	Семенова Е.Д.	фикс.	27 628,15	27 628,15	3 591,66	24 036,49
5	Яковлева М.Н.	почас.	302,00	48 320,00	6 281,60	42 038,40
Всего:				178 922,15	23 259,88	155 662,27

Контрольные вопросы

1. Перечислите три принципа ООП.
2. Поясните на примерах, что такое инкапсуляция, наследование, полиморфизм.
3. Что такое свойства классов? Чем они отличаются от полей и методов?
4. Что такое автоматически определяемые свойства?
5. Какие вы знаете области видимости? Чем они отличаются?
6. Как указать класс-предок в C#?
7. Зачем нужны операторы `is` и `as`?
8. Что такое абстрактный класс и абстрактный метод?
9. Что такое конструктор? Сколько конструкторов может быть у класса?
10. Что такое переопределение методов? Какие ключевые слова нужно использовать?
11. Что такое перегрузка методов?
12. Что означают ключевые слова `this` и `base`?
13. Как выполняется генерация случайных чисел?

5. СОЗДАНИЕ ПРИЛОЖЕНИЙ С ГРАФИЧЕСКИМ ПОЛЬЗОВАТЕЛЬСКИМ ИНТЕРФЕЙСОМ. WINDOWS FORMS

Пример 1. Простейшее оконное приложение

Создать простейшее оконное приложение - аналог консольного HelloUser. Разместить на форме кнопку и поле для ввода имени пользователя.

При вводе имени пользователя на форме должна появляться надпись с приветствием. При нажатии на кнопку должно возникнуть окно со всплывающим сообщением.

Создание оконного приложения

Это будет наша первая программа, выполненная не в виде консоли, а в виде оконного приложения Windows.

При создании нового проекта выберите тип **Windows Forms Application** (Приложение Windows Forms), имя проекта - "SimpleForm" (рис. 5.1).

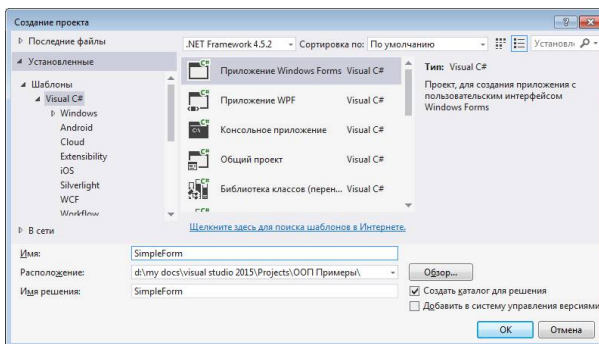


Рис. 5.1. Создание приложения Windows Forms Application

После создания проекта вы увидите не программный код, как раньше, а пустую форму, на которой можно размещать кнопки, текст, поля для ввода, флажки, переключатели, выпадающие списки и многие другие **элементы управления**.

Все их можно найти на **Панели элементов (Toolbox)** в правой части экрана. Также нам потребуются окна **Обозреватель решений (Solu-**

tion Explorer) и Свойства (Properties), как показано на рис. 5.2. Если эти окна у вас не отображаются, включите их через меню Вид (View) и разместите, как на скриншоте. Закрепить окна, чтобы они не сворачивались, можно кнопкой с изображением канцелярской кнопки.

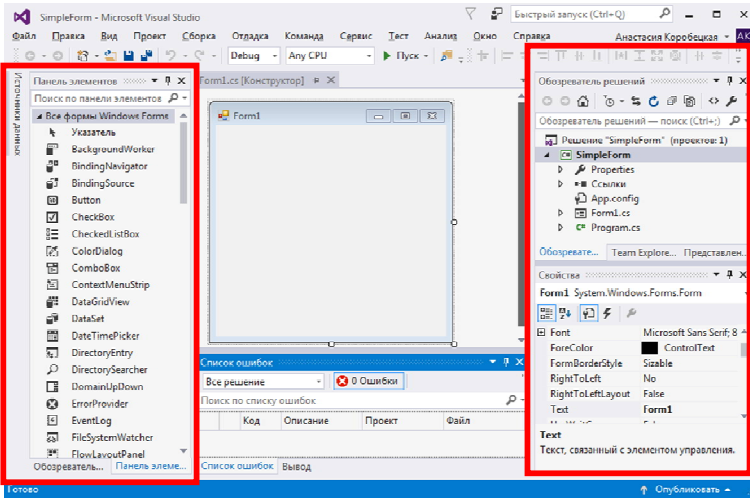



Рис. 5.2. Панель элементов, обозреватель решений и окно свойств

Все настройки формы и ее элементов управления находятся в окне обозревателя свойств. Они собраны в тематические группы, которые можно сворачивать и разворачивать. Снизу выводятся пояснения к выделенному свойству.

Попробуйте разместить на форме кнопку ( **Button**), как показано на рис. 5.3. По умолчанию на ней размещен текст “button1” и называется она “button1” (имя кнопки и текст на кнопке - это разные вещи).

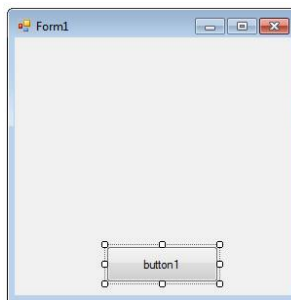


Рис. 5.3. Размещение кнопки на форме

Выделите созданную кнопку и найдите ее свойство Text (надпись) в группе "Внешний вид". Впишите туда "Нажми меня", а в свойство (Name) из группы "Разработка" впишите buttonClickMe (рис. 5.4).



Рис. 5.4. Настройки свойств кнопки

В дальнейшем перечень свойств, которые нужно изменить, будем записывать так (кавычки писать не нужно):

Text = "Нажми меня"

(Name) = buttonClickMe

Примечание. Изменять имена (Name) компонентов - это очень важно. Не используйте стандартные имена button1, textBox2 и т.д. Уже после третьей кнопки вы забудете, какая из них за что отвечает. А при использовании "говорящих" имен сразу понятно, что buttonClickMe - это кнопка "Нажми меня", а textBoxUserName - это поле для ввода имени пользователя. Имена элементов формы задаются в camelStyle.

Чтобы переименовать форму, нужно изменить имя ее файла в Обозревателе решений и подтвердить изменения.

При этом имя формы изменится, а надпись в заголовке останется Form1. Задайте надпись в заголовке формы:

Text = "Простое оконное приложение"

Установим размер кнопки равным 200x40 пикселей. Ширина (Width) и высота (Height) могут быть скрыты в общей группе Size (рис. 5.5).

Size.Width = 200

Size.Height = 40

Size	115; 37
Width	115
Height	37

Рис. 5.5. Свойство Size содержит два вложенных свойства Width и Height

Цвет кнопки сделаем желтым (только для тренировки, на практике делать разноцветные кнопки не рекомендуется):

BackColor = Yellow

Шрифт - полужирный, размер 11:

Font.Bold = True

Font.Size = 11

В результате получим кнопку вида, показанного на рис. 5.6.

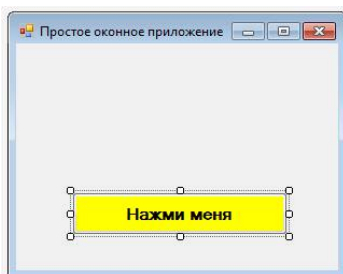


Рис. 5.6. Вид кнопки после применения настроек

Разместите на форме еще один объект - метку (A Label) для отображения текста. Задайте ей имя и удалите текст:

(Name) = labelHello

Text = ""

В результате метка станет "невидимой", но ее все еще можно выделить на форме (рис. 5.7).

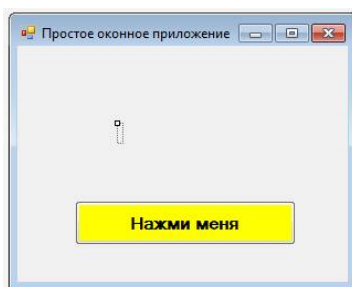


Рис. 5.7. Выделение метки, не содержащей текста

Если вы потеряли какой-то объект на форме и не можете выделить его, все элементы можно найти вверху обозревателя свойств (рис. 5.8).

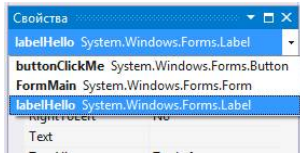



Рис. 5.8. Выбор элемента из списка

Поместите labelHello в середине формы. Зажав кнопку Ctrl, вы можете включить привязку к краям и центрам элементов, чтобы сделать позиционирование более точным.

Третьим разместите на форме поле для ввода текста  TextBox (рис. 5.9).

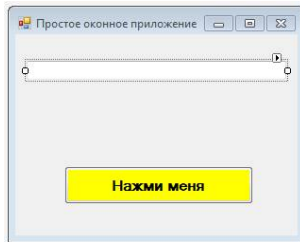


Рис. 5.9. Размещение поля для ввода текста на форме

Задайте ему имя:

(Name) = textBoxUserName

Над полем ввода разместите еще одну метку (рис. 5.10):

(Name) = labelUserName

Text = "Как Вас зовут?"

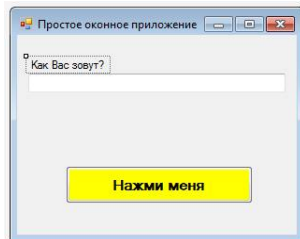


Рис. 5.10. Размещение метки с пояснением на форме

События

Попробуем добавить программный код. Нужно понимать, что при работе с формой порядок выполнения программы отличается от консоли.



В консоли у вас сразу запускался метод Main и выполнялся написанный в нем код.

В оконном приложении метод Main существует, но мы не работаем прямо с ним. Он только создает нашу форму и выводит ее на экран, делая это автоматически, вручную писать в него код не нужно. Дальнейшая работа приложения происходит в ответ на **события (Events)**.

Событием является любое действие пользователя, подхваченное системой: нажатие на кнопку, перемещение курсора, ввод данных и т.п. Источником событий может быть не только пользователь, но и другие программы, операционная система, сетевые подключения, таймеры и др.

Само по себе событие ничего не делает. К нему необходимо привязать **обработчик** - метод, который работает при возникновении события.

Доступные события, как и свойства и методы, являются частью описания класса. У разных объектов события отличаются. Например, и по кнопке, и по форме, и по текстовому полю можно кликнуть (Click) и отловить наведение курсора (Hover). Но только в поле можно изменить текст (TextChanged). Только у формы есть событие при ее отображении на экране (Show).

Все доступные события можно просмотреть в окне свойств, кликнув по кнопке с молнией . Кнопка с гаечным ключом и листом бумаги  вновь отобразит свойства (рис. 5.11).

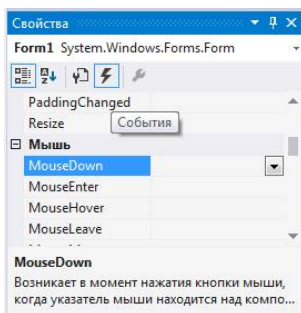


Рис. 5.11. Переключение на список событий в окне "Свойства"

Добавим обработку события "нажата кнопка buttonClickMe". Дважды щелкните левой кнопкой мыши по кнопке на форме либо по ее событию Click.

VisualStudio переключится в режим редактирования кода и добавит новый метод `buttonClickMe_Click`, который будет привязан к соответствующему событию (рис. 5.12).

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void buttonClickMe_Click(object sender, EventArgs e)
    {
    }
}
```

Рис. 5.12. Автоматически созданный обработчик события Click

Исходный код формы также можно открыть клавишей **F7**.

Удаление обработчика события

Если вы случайно создали ненужный обработчик события, удалять его нужно в два шага:

- 1) убрать привязку к событию,
- 2) стереть обработчик из исходного кода.

Допустим, мы случайно создали не обработчик события Click для кнопки, а обработчик загрузки Load для формы (именно он создается при двойном клике по форме).

```
private void Form1_Load(object sender, EventArgs e)
{
}
}
```

Прежде всего, найдите событие в окне свойств. Кликните по нему правой кнопкой и выберите пункт "Сброс" (рис. 5.13). В результате привязка метода к событию удалится. Если вы попытаетесь стереть текст метода, не отвязав его от события, то это приведет к плачевному результату - форма вообще перестанет открываться.

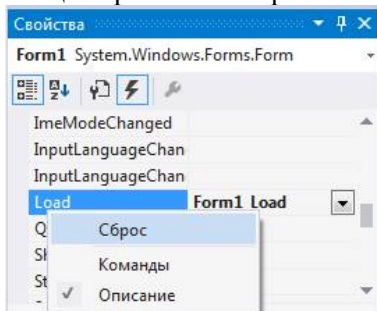


Рис. 5.13. Удаление обработчика события

Теперь можно переключиться в исходный код (F7) и стереть метод `Form1_Load`.

Обработчик нажатия кнопки

Вернемся к созданному нами методу `buttonClickMe_Click`.

? Какой смысл имеют параметры `sender` и `e`, которые есть у обработчика? В наших простых примерах они не понадобятся, но на практике будут необходимы.

Весь код, который должен сработать при нажатии на кнопку, пишем внутри метода `buttonClickMe_Click`.

Например, выведем надпись в нашу метку `labelHello`. Для этого нужно присвоить значение ее свойству `Text`.

```
--
20     private void buttonClickMe_Click(object sender, EventArgs e)
21     {
22         labelHello.Text = "Спасибо, что нажали!";
23     }
```

Запустите программу и проверьте работу нашей кнопки (рис. 5.14).

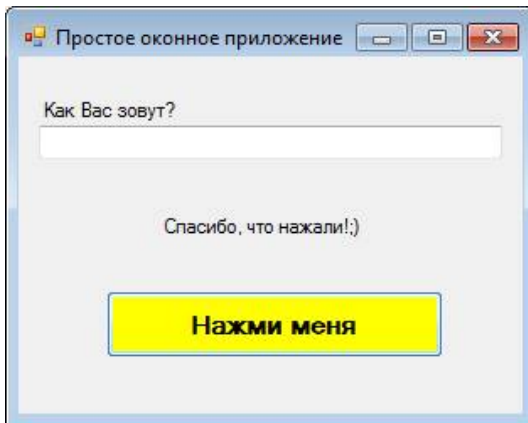


Рис. 5.14. Результат нажатия на кнопку

Но по заданию нас просили вывести всплывающее сообщение. Это делается с помощью `MessageBox.Show` (рис. 5.15).

```
20     private void buttonClickMe_Click(object sender, EventArgs e)
21     {
22         MessageBox.Show("Спасибо, что нажали!");
23     }
```

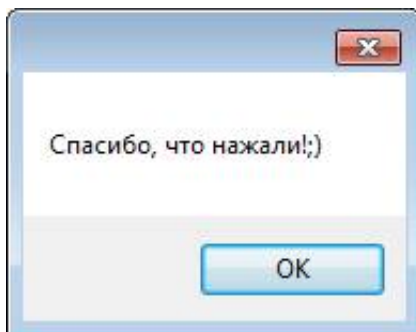


Рис. 5.15. Вывод всплывающего сообщения

У `MessageBox.Show` есть несколько дополнительных параметров - иконка, список отображаемых кнопок, заголовков окна и др.

Обработчик ввода текста

При наборе имени пользователя в метке `labelHello` должно выводиться приветствие.

Создайте обработчик метода `TextChanged` для `textBoxUserName`.

```
25 private void textBoxUserName_TextChanged(object sender, EventArgs e)
26 {
27     labelHello.Text = "Здравствуйте, " + textBoxUserName.Text + "!";
28 }
```

Обратите внимание, что присваивать надо в свойство `Text`, а не в саму метку.

Обработчик будет срабатывать после ввода каждой буквы, а также при удалении текста, вставке из буфера обмена (рис. 5.16).

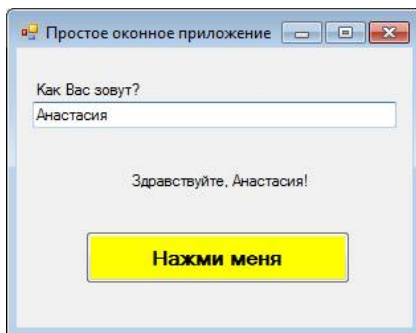


Рис. 5.16. Вывод имени пользователя в метку

Обработчик нажатия на метку

Последний штрих - сделаем так, чтобы при нажатии на кнопку labelUserName курсор установился в textBoxUserName. К сожалению, такого свойства у Label нет, нужно добавить обработчик события Click.

```
30 private void labelUserName_Click(object sender, EventArgs e)
31 {
32     textBoxUserName.Focus();
33 }
```

Запустите приложение и убедитесь, что при нажатии на метку фокус устанавливается в textBoxUserName.

Пример 2. Текстовый редактор

Создать приложение - аналог Блокнота Windows. Приложение должно поддерживать ввод и редактирование текста в окне, операции с буфером обмена, сохранение и открытие файла, создание нового файла.

Следует запомнить, в какой файл был сохранен текст, и не запрашивать путь к файлу при повторном сохранении. Имя файла нужно вывести в заголовке формы.

Необходимо отслеживать, были ли сделаны изменения после последнего сохранения. Если были, то вывести звездочку в заголовке формы и спросить, не нужно ли сохранить текст при создании нового файла, открытии файла, закрытии приложения.

Предусмотреть кнопку выхода и форму "О программе".

Создание форм. Размещение элементов

Создайте новое оконное приложение под именем "MyNotepad" (Мой блокнот).

Переименуйте созданную **форму** в FormNotepad. Задайте ей заголовок "Мой блокнот". Найдите в Интернете подходящую иконку в формате ICO и назначьте ее форме и проекту в целом. Сделайте форму побольше. Результат и настройки формы показаны на рис. 5.17 и в табл. 5.1.

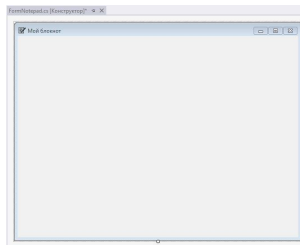


Рис. 5.17. Внешний вид главной формы приложения "Мой блокнот"

Настройки главной формы приложения "Мой блокнот"

Группа	Свойство	Значение	Примечание
Разработка	(Name)	FormNotepad	Переименовать файл в "Обозревателе решений"
Внешний вид	Text	Мой блокнот	
Стиль окна	Icon	Выбрать файл иконки в формате .ico	Найти подходящую иконку в Интернете и сохранить в папку с проектом
Макет	Size	640; 480	

Также назначьте **иконку** самому **приложению**. Откройте свойства проекта (вкладка "Приложение"), выберете "Значок:", как показано на рис. 5.18. В результате иконка добавится в проект в "Обозревателе решений" (рис. 5.19).

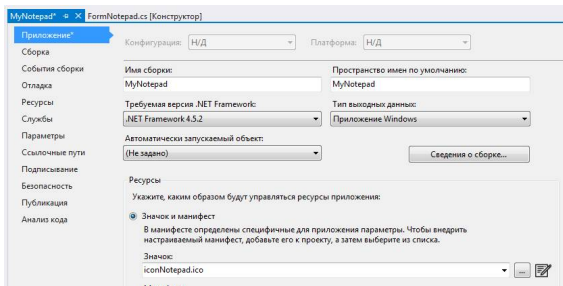


Рис. 5.18. Выбор иконки приложения "Мой блокнот"

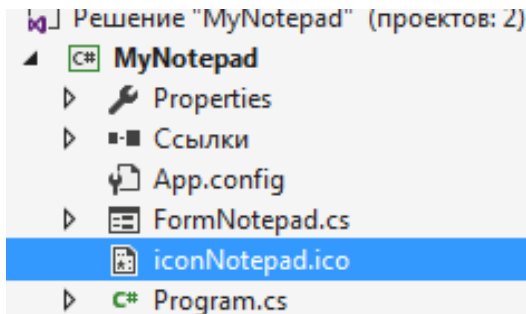
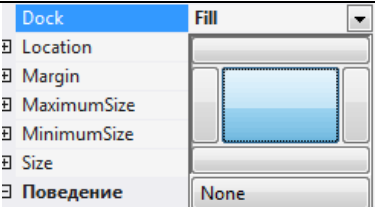



Рис. 5.19. Иконка приложения в "Обозревателе решений"

Разместим на форме поле для ввода текста. Для многострочного текста в VisualStudio используется тот же TextBox, что и для однострочного, нужно только установить свойство Multiline в True. Также необходимо, чтобы текстовое поле занимало всю форму - за это отвечает свойство Dock. Все значения свойств показаны в табл. 5.2.

Таблица 5.2

Свойства поля для ввода текста в приложении "Мой блокнот"

Группа	Свойство	Значение	Примечание
Разработка	(Name)	textBoxNote- pad	
Внешний вид	Text	""	
	Scrollbars	Both	Наличие обеих полос прокрутки - вертикальной и горизонтальной
Поведение	Multiline	True	
Макет	Dock	Fill	

Добавьте на форму **главное меню** (элемент  MenuStrip из группы "Меню и панели инструментов"). Создайте структуру меню, показанную на рис. 5.20.

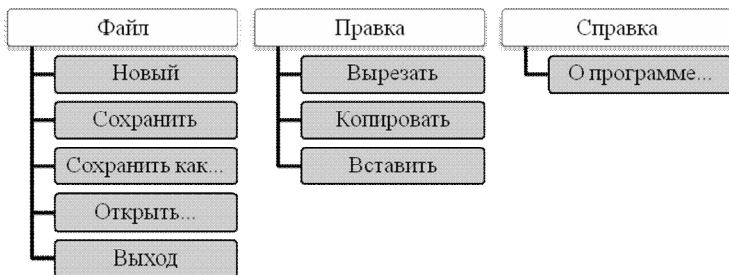


Рис. 5.20. Структура меню приложения "Мой блокнот"

Добавьте разделитель (Separator) перед пунктом "Выход" (рис. 5.21).

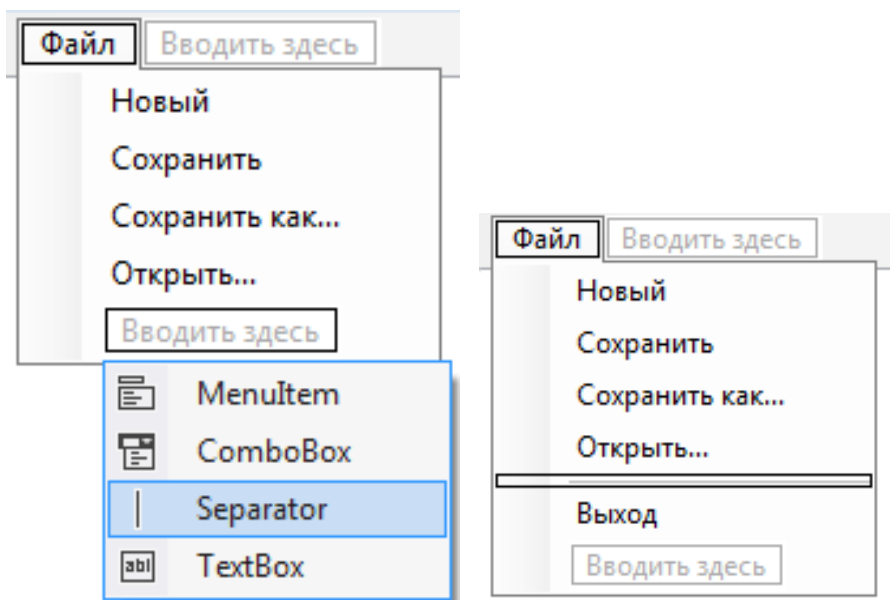


Рис. 5.21. Вставка разделителя в меню

Задайте **пунктам меню** имена и назначьте горячие клавиши (свойство `ShortcutKeys` в группе "Прочее"), как показано в табл. 5.3 и на рис. 5.22. По умолчанию имена будут содержать русские буквы, что может привести к проблемам с кодировкой.

Таблица 5.3

Названия и горячие клавиши пунктов меню

Пункт	(Name)	ShortcutKeys
Новый	<code>toolStripMenuItemNew</code>	<code>Ctrl+N</code>
Сохранить	<code>toolStripMenuItemSave</code>	<code>Ctrl+S</code>
Сохранить как...	<code>toolStripMenuItemSaveAs</code>	<code>Ctrl+Alt+S</code>
Открыть...	<code>toolStripMenuItemOpen</code>	<code>Ctrl+O</code>
Выход	<code>toolStripMenuItemExit</code>	
Вырезать	<code>toolStripMenuItemCut</code>	<code>Ctrl+X</code>
Копировать	<code>toolStripMenuItemCopy</code>	<code>Ctrl+C</code>
Вставить	<code>toolStripMenuItemPaste</code>	<code>Ctrl+V</code>
О программе...	<code>toolStripMenuItemAbout</code>	

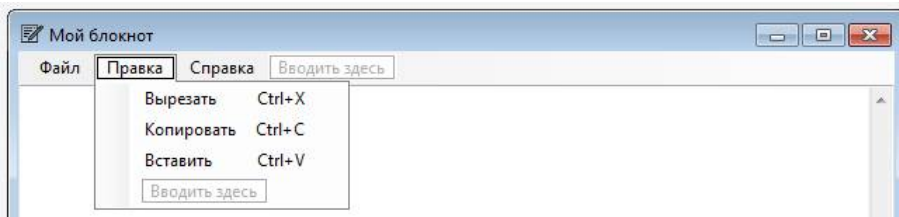


Рис. 5.22. Вид пунктов меню с настроенными горячими клавишами

Чтобы выбрать путь для сохранения и открытия файлов, добавьте соответствующие диалоги из группы элементов "Диалоговые окна". Добавленные диалоги отобразятся внизу под формой, как и MenuStrip.

Задайте обоим диалогам фильтр для выбора текстовых файлов (свойство Filter в группе свойств "Поведение"):

`Filter = "Текстовые документы|*.txt|Все файлы|*.*"`

Диалогу сохранения задайте расширение по умолчанию (оно будет добавлено к имени файла, если пользователь не укажет расширение):

`DefaultExt = ".txt"`

Еще нам потребуется вторая форма "О программе". Создайте новую форму через меню "Проект" - "Добавить форму Windows...". Назовите ее "FormAbout". Самостоятельно разместите на форме элементы, чтобы она выглядела как показано на рис. 5.23 и в табл. 5.4 (укажите свои ФИО).

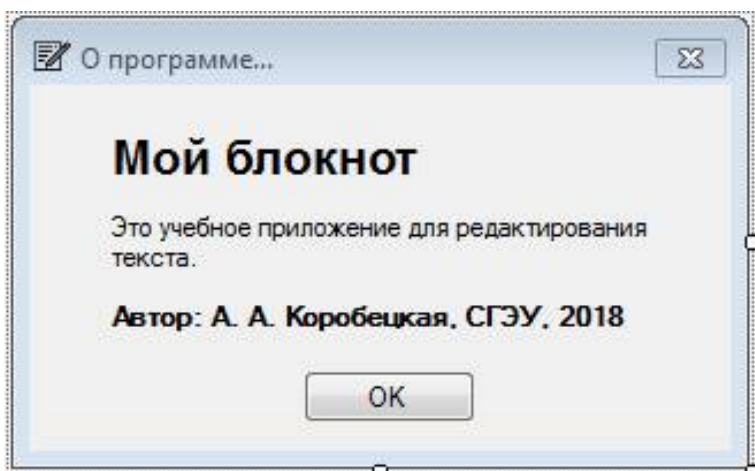


Рис. 5.23. Внешний вид формы FormAbout

Свойства формы FormAbout

Группа	Свойство	Значение	Примечание
Внешний вид	FormBorderStyle	FixedDialog	Нельзя будет изменять размер формы
	Text	"О программе"	
Макет	Size	320; 200	
	StartPosition	ParentCenter	Будет отображаться по центру родительского окна
Стиль окна	Icon	Файл иконки в формате .ico	Та же, что и для главной формы
	MaximizeBox	False	Отключает кнопки "Свернуть" и "Развернуть" на рамке формы
	MinimizeBox	False	
	ShowInTaskbar	False	Форма не будет отображаться в панели задач Windows

Чтобы разместить в Label многострочный текст, отключите у нее свойство AutoSize.

Чтобы выровнять кнопку по центру, выделите ее и выберите в меню "Формат" - "По центру формы" - "По горизонтали".

На этом разработку пользовательского интерфейса завершим.

Обработчики для пунктов меню

Перейдем к коду для обработки событий формы.

Меню "Правка", "Выход"

Начнем с наиболее простой части кода - меню "Правка". Все три действия "Вырезать", "Копировать" и "Вставить" уже встроены в методы TextBox. Добавьте пунктам меню обработчики события Click:

```

20 private void toolStripMenuItemCut_Click(object sender, EventArgs e)
21 {
22     textBoxNotepad.Cut();
23 }
24
25 private void toolStripMenuItemCopy_Click(object sender, EventArgs e)
26 {
27     textBoxNotepad.Copy();
28 }
29
30 private void toolStripMenuItemPaste_Click(object sender, EventArgs e)
31 {
32     textBoxNotepad.Paste();
33 }

```

К пункту меню "**Выход**" нужно привязать команду Close() - закрытие формы. Поскольку FormNotepad - главная форма приложения, ее закрытие приводит и к закрытию всего приложения.

```
35     private void toolStripMenuItemExit_Click(object sender, EventArgs e)
36     {
37         Close();
38     }
```

Запустите приложение и убедитесь, что все четыре пункта меню работают, в том числе по горячим клавишам.

О программе

Следующий шаг - форма "**О программе**". Пункт меню должен показать форму. Для этого, как и любой объект, ее нужно сначала создать из класса FormAbout, а затем вызвать метод Show() или ShowDialog(), для чего можно записать ее в переменную, а потом вызвать метод для этой переменной.

```
40     private void toolStripMenuItemAbout_Click(object sender, EventArgs e)
41     {
42         FormAbout f = new FormAbout();
43         f.ShowDialog();
44     }
```

Но поскольку больше нигде мы к этой форме обращаться не будем, можно записать в одну строку:

```
40     private void toolStripMenuItemAbout_Click(object sender, EventArgs e)
41     {
42         (new FormAbout()).ShowDialog(this);
43     }
```

При использовании метода ShowDialog(), в отличие от Show():

- 1) нельзя переключиться в главное окно, пока диалоговое окно не будет закрыто;
- 2) можно узнать результат вызова диалога - ОК, отмена, применить и другие значения собраны в DialogResult.

Указывая this в параметрах, мы сообщаем FormAbout, что ее вызвала форма FormNotepad. Без этого не будет работать свойство StartPosition = ParentCenter.

На самой форме "О программе" не забудьте написать обработчик для кнопки "ОК".

```

20     private void buttonOK_Click(object sender, EventArgs e)
21     {
22         Close();
23     }

```

Протестируйте работу приложения. Убедитесь, что при открытии формы "О программе":

- 1) она размещается по центру главной формы;
- 2) ее размер не изменяется;
- 3) нельзя переключиться на главную форму;
- 4) кнопка "ОК" работает.

Меню "Файл"

Остались пункты меню для работы с файлами - "Новый", "Сохранить", "Сохранить как", "Открыть". Это более сложные действия, где одной строчкой кода не обойтись. В таком случае лучше написать приватные методы формы, которые будут выполнять соответствующие действия, а из обработчиков событий только вызывать эти методы.

Пока что создадим три пустых void-метода NewFile, OpenFile и SaveFile. Для открытия и сохранения файла необходимо передать имя и путь к файлу fileName в качестве параметра.

Примечание. Общепринятым является отмечать незавершенные фрагменты кода комментарием TO DO. В нашем небольшом проекте запутаться сложно, а в программе на несколько десятков тысяч строк кода поиск по TO DO позволяет убедиться, что вы ничего не забыли.

```

20     private void NewFile()
21     {
22         // TO DO
23     }
24
25     private void OpenFile(string fileName)
26     {
27         // TO DO
28     }
29
30     private void SaveFile(string fileName)
31     {
32         // TO DO
33     }

```

Создайте обработчики событий.

Начнем с "Сохранить как..." - здесь нужно отобразить диалог сохранения, и если пользователь подтвердил выбор файла, то вызвать SaveFile.

```

59     private void toolStripMenuItemSaveAs_Click(object sender, EventArgs e)
60     {
61         if (saveFileDialog1.ShowDialog() == DialogResult.OK)
62         {
63             SaveFile(saveFileDialog1.FileName);
64         }
65     }

```

Это общий подход к работе с диалоговыми окнами: отобразить методом ShowDialog, проверить результат DialogResult, выполнить действие.

Но конкретно для SaveFileDialog и ShowFileDialog присутствует событие FileOk, которое возникает при успешном выборе файла. Поэтому лучше написать обработчик этого события, а в клике по пункту "Сохранить как..." оставить только одну строчку.

```

59     private void toolStripMenuItemSaveAs_Click(object sender, EventArgs e)
60     {
61         saveFileDialog1.ShowDialog();
62     }
63
64     private void saveFileDialog1_FileOk(object sender, CancelEventArgs e)
65     {
66         SaveFile(saveFileDialog1.FileName);
67     }

```

Пункт "Сохранить" отличается от "Сохранить как..." тем, что если уже был выбран путь к файлу, то нужно сохранить его туда же, а если еще не был, то отобразить диалог сохранения. То есть нам нужно как-то проверить, был ли сохранен файл и куда. Для этого заведем у формы поле и свойство FileName, куда будем записывать имя файла при каждом сохранении.

```

10     /
19
20     string fileName = "";
21     public string FileName
22     {
23         get { return fileName; }
24         set { fileName = value; }
25     }
26
27     private bool FileChanged()

```

? Подумайте, почему обязательно нужно создавать поле для сохранения имени файла. Почему нельзя воспользоваться saveFileDialog1.FileName - оно ведь тоже пусто, пока не был выбран файл? К каким ошибкам это может привести?

Тогда обработчик события для пункта "Сохранить" будет выглядеть так:

```

83     private void toolStripMenuItemSave_Click(object sender, EventArgs e)
84     {
85         if (FileName == "")
86         {
87             saveFileDialog1.ShowDialog();
88         }
89         else
90         {
91             SaveFile(FileName);
92         }
93     }

```

Теперь создадим обработчик для пункта "Новый". При создании нового файла нужно проверить, нет ли изменений, которые нужно сохранить.

Добавим в FormNotepad еще одно свойство, в котором будет отмечаться наличие изменений в файле.

```

27         bool fileChanged = false;
28         public bool FileChanged
29         {
30             get { return fileChanged; }
31             set { fileChanged = value; }
32         }

```

Отследить изменения в тексте можно по событию TextChanged у textBoxNotepad.

```

140     private void textBoxNotepad_TextChanged(object sender, EventArgs e)
141     {
142         FileChanged = true;
143     }

```

Теперь напишем заготовку для обработчика клика по пункту "Новый":

```

97     private void toolStripMenuItemNew_Click(object sender, EventArgs e)
98     {
99         if (FileChanged)
100         {
101             // TO DO спросить у пользователя, хочет ли он сохранить файл
102         }
103         NewFile();
104     }

```

MessageBox

Если изменения есть, то нужно задать вопрос пользователю, хочет ли он сохранить файл, т.е. необходимо вывести окно, аналогичное показанному на рис. 5.24.

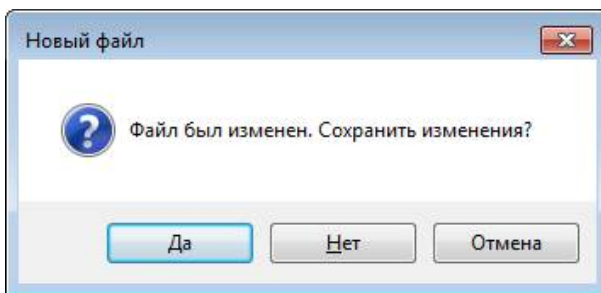


Рис. 5.24. Окно подтверждения создания нового файла

В предыдущем примере мы уже использовали `MessageBox.Show()` для вывода сообщения, но там была только кнопка "ОК". Метод `MessageBox.Show()` имеет много (всего 21) перегрузок с различными настройками окна сообщения. Нас интересует следующая:

```
DialogResult Show(string text, string caption,
MessageBoxButtons buttons, MessageBoxIcon icon)
```

Параметры:

- `text` - текст сообщения (у нас "Файл был изменен. Сохранить изменения?");
- `caption` - заголовок окна сообщения ("Новый файл");
- `buttons` - какие кнопки отображать (Да, Нет, Отмена - `MessageBoxButtons.YesNoCancel`);
- `icon` - иконка слева от сообщения (знак вопроса, `MessageBoxIcon.Question`).

Метод `Show()` возвращает нажатую кнопку. С помощью `switch` можно прореагировать на каждый вариант:

- `Yes` - вызвать метод сохранения файла, а потом создать новый;
- `No` - создать новый, не сохраняя;
- `Cancel` - не сохранять и не создавать новый (прервать обработчик).

Таким образом, можно дописать обработчик и убрать метку "TO DO":

```
97 private void toolStripMenuItemNew_Click(object sender, EventArgs e)
98 {
99     if (FileChanged)
100     {
101         switch (MessageBox.Show("Файл был изменен. Сохранить изменения?",
102             "Новый файл", MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question))
103         {
104             case DialogResult.Yes: toolStripMenuItemSave_Click(sender, e); break;
105             case DialogResult.No: break;
106             case DialogResult.Cancel: return;
107         }
108     }
109     NewFile();
110 }
```

Аналогично напишем открытие файла.

```
112     private void toolStripMenuItemOpen_Click(object sender, EventArgs e)
113     {
114         if (FileChanged)
115         {
116             switch (MessageBox.Show("Файл был изменен. Сохранить изменения?",
117                 "Открыть файл", MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question))
118             {
119                 case DialogResult.Yes: toolStripMenuItemSave_Click(sender, e); break;
120                 case DialogResult.No: break;
121                 case DialogResult.Cancel: return;
122             }
123         }
124         openFileDialog1.ShowDialog();
125     }
126
127     private void openFileDialog1_FileOk(object sender, CancelEventArgs e)
128     {
129         OpenFile(openFileDialog1.FileName);
130     }
```

Запустите приложение и убедитесь, что окно сообщения, диалоги сохранения и загрузки отображаются, хотя файлы еще не работают.

Декомпозиция программы

Как видите, мы пока что реализовали логику работы пунктов меню, не отвлекаясь и не задумываясь о том, как работать с файлами.

Выделение небольших коротких методов или свойств под каждую маленькую задачу (**декомпозиция программы**) сильно облегчает разработку и последующее развитие приложения. Старайтесь следовать этому принципу.

Когда нужно выделить отдельный метод/свойство?

НУЖНО:

- если хотя бы три строчки кода повторяются в программе не менее 2 раз;
- вы можете выразить смысл фрагмента кода простой фразой, например, "вывести приветствие", "сохранить результат в файл", "задать ширину колонки", "соединение установлено?" (это и будет имя метода/свойства);
- один метод по длине не уместится на экране (больше 20-30 строк);
- фрагменты кода решают разные задачи;
- фрагменты кода пишутся разными разработчиками;
- данная задача в будущем будет/может решаться как-то иначе, и нужно будет изменить фрагмент кода, не сломав остальную программу.

НЕ НУЖНО:

- если код содержит одну-две строчки и повторяется только 1 раз, причем шансы, что строчек/повторов станет больше, крайне низки;

- в приложении критично время выполнения или занимаемый объем памяти, на счету каждый байт и каждый такт процессора (вызов метода - довольно затратная операция);

- вы делаете прототип на коленке за 10 минут, а потом с нуля напишете, как положено (только если вы уверены, что не получится по принципу "нет ничего более вечного, чем временное");

- никто никогда больше не откроет этот код, включая вас самих;

- приложение очень простое (гораздо проще нашего блокнота).

Помните: *код пишут 1 раз, а читают много*. Выделение методов сильно облегчает чтение кода и сокращает количество необходимых комментариев, так как название метода само служит комментарием.

? Нужно ли выделить еще какой-то метод в нашем коде?

Работа с файлами

Теперь реализуем функционал методов `NewFile`, `OpenFile`, `SaveFile`.

В `NewFile` нужно очистить `textBoxNotepad`, отметить, что в файле еще не было изменений и очистить имя файла.

```
34 private void NewFile()
35 {
36     FileChanged = false;
37     FileName = "";
38     textBoxNotepad.Clear();
39 }
```

Для открытия и сохранения файла нам потребуется пространство имен `System.IO` (Input/Output - ввод/вывод). Добавьте его в список `using` в начале файла.

```
7 using System.Windows.Forms;
10 using System.IO;
```

Для чтения данных из файла (или из другого источника, например, из Сети) служит поток чтения `StreamReader`. Необходимо создать поток (он сразу подключится к указанному файлу), выгрузить весь текст до конца файла в `textBoxNotepad` и закрыть поток (иначе файл так и останется открытым и заблокированным, его нельзя будет удалить, переместить и т.д.). Так же, как и при создании нового файла, нужно сбросить отметку о наличии изменений в файле.

```

42     private void OpenFile(string fileName)
43     {
44         // TO DO
45         StreamReader file = new StreamReader(fileName);
46         textBoxNotepad.Text = file.ReadToEnd();
47         file.Close();
48         FileChanged = false;
49         FileName = fileName;
50     }

```

Обработка исключений *try-catch-finally*

Нужно помнить, что действия с файлами - это уязвимая часть кода. Пока мы вызываем метод, с файлом может что-то случиться (удалили, переименовали, изменили права доступа). Если в процессе открытия или чтения файла возникнет какая-то ошибка, то будет **выброшено исключение (exception)** и программа вылетит или начнет пугать пользователя непонятными сообщениями.

Сами по себе исключения - это неплохо, это просто способ сообщить программисту, что что-то пошло не так. А наша задача - поймать и обработать исключение, прежде чем оно дойдет до пользователя.

Поэтому необходимо обернуть код в конструкцию **try-catch-finally** (попытайся - поймай - в конце).

В блок try помещается потенциально опасный код, который может вызвать исключения. Если исключения не будет, он выполнится как обычно. А если будет, то блок try прервется (не будет выполнен до конца) и программа перейдет к catch и finally.

В блок catch можно попасть, только если возникло исключение. Если исключения не было, он будет пропущен. Еще в нем можно проверить тип исключения, чтобы понять, какая именно ошибка произошла.

В блок finally мы попадем всегда, было исключение или нет. Здесь размещаются важные действия, которые обязательно нужно выполнить.

Один из блоков catch или finally можно пропустить.

Таким образом, нам нужно:

- попытаться открыть и прочитать файл в try;
 - при успешном чтении (в конце try) сбросить FileChanged и запомнить FileName;
 - при исключении (в catch) вывести сообщение об ошибке;
 - в конце обязательно (finally) закрыть файл, если он был открыт.
- Все вместе будет выглядеть так:

```

42     private void OpenFile(string fileName)
43     {
44         StreamReader file = null;
45         try
46         {
47             file = new StreamReader(fileName);
48             textBoxNotepad.Text = file.ReadToEnd();
49             FileChanged = false;
50             FileName = fileName;
51         }
52         catch
53         {
54             MessageBox.Show("Возникла ошибка при открытии файла!", "Ошибка",
55                 MessageBoxButtons.OK, MessageBoxIcon.Error);
56         }
57         finally
58         {
59             if (file != null)
60             {
61                 file.Close();
62             }
63         }
64     }

```

Сохранение файла выполняется аналогично, только для чтения применяется класс `StreamWriter`.

```

66     private void SaveFile(string fileName)
67     {
68         StreamWriter file = null;
69         try
70         {
71             file = new StreamWriter(fileName);
72             file.Write(textBoxNotepad.Text);
73             FileChanged = false;
74             FileName = fileName;
75         }
76         catch
77         {
78             MessageBox.Show("Возникла ошибка при сохранении файла!", "Ошибка",
79                 MessageBoxButtons.OK, MessageBoxIcon.Error);
80         }
81         finally
82         {
83             if (file != null)
84             {
85                 file.Close();
86             }
87         }
88     }

```

Запустите программу и убедитесь, что сохранение и загрузка работают корректно.

Отслеживание изменений в заголовке формы

Мы не выполнили еще два пункта из задания: в заголовке формы должно отображаться имя файла (если оно есть) и звездочка, если есть несохраненные изменения.

Иными словами, нам нужно изменить заголовок формы при изменении свойств `FileName` и `FileChanged`. Именно для этого и нужны

сеттеры. Для обоих свойств добавим в set вызов метода `ChangeFormCaption()`, который будет формировать нужную строку в заголовке (`Caption`) формы.

```
21     string fileName = "";
22     public string FileName
23     {
24         get { return fileName; }
25         set
26         {
27             fileName = value;
28             ChangeFormCaption();
29         }
30     }
31
32     bool fileChanged = false;
33     public bool FileChanged
34     {
35         get { return fileChanged; }
36         set
37         {
38             fileChanged = value;
39             ChangeFormCaption();
40         }
41     }
```

Осталось написать сам метод:

```
43     private void ChangeFormCaption()
44     {
45         Text = "Мой блокнот";
46         if (FileName != "")
47         {
48             Text += " - " + FileName.Substring(FileName.LastIndexOf("\\") + 1);
49         }
50         if (FileChanged)
51         {
52             Text += "***";
53         }
54     }
```

Напомним, что мы пишем метод формы `FormNotepad`, поэтому свойство `Text` относится именно к ней.

Чтобы вывести имя файла из полного пути в `FileName`, необходимо найти последний слеш (`LastIndexOf`) и обрезать строку (`Substring`), начиная со следующего символа.

? Помните, почему нужно писать `\\` вместо `\`

Запустите приложение и убедитесь, что при вводе текста в имени формы отображается звездочка, при сохранении изменений звездочка

пропадает и отображается имя файла, а при дальнейших изменениях снова появляется (рис. 5.25, 5.26, 5.27).

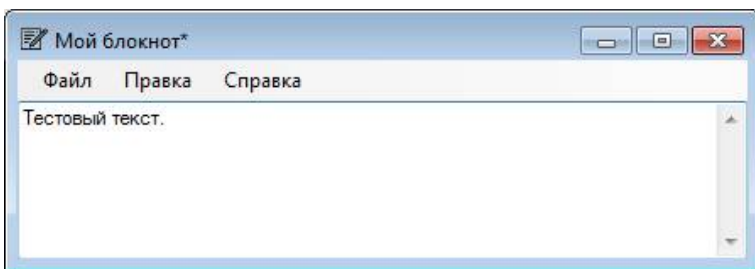


Рис. 5.25. Отображение звездочки в заголовке формы при изменении текста

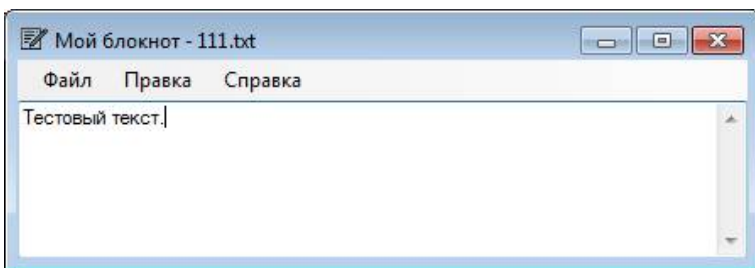


Рис. 5.26. Отображение имени файла без звездочки после сохранения

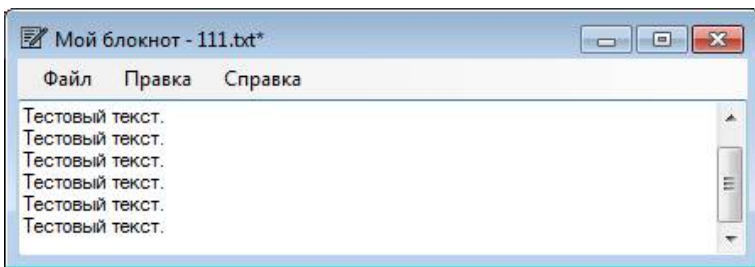


Рис. 5.27. Отображение имени файла со звездочкой после сохранения и редактирования текста

Настройка шрифта

Самостоятельно добавьте на форму диалог для выбора шрифта `FontDialog` и добавьте в меню пункты "Вид" - "Шрифт...". Напишите код для установки выбранного шрифта (свойство `Font`) всему `textBoxNotepad`.

Задачи для самостоятельного решения

Задача 1. Просмотрщик изображений

Создайте приложение Windows Forms, позволяющее открывать и просматривать графические файлы. Используйте PictureBox для размещения рисунков на форме. Добавьте на форму кнопки и необходимые диалоги:

- для загрузки изображения из файла;
- очистки изображения;
- выбора цвета фона.

В нижней части формы должны отображаться сведения об открытом файле: формат, ширина и высота в пикселях, размер файла в МБ/кБ/байтах (подходящую единицу измерения выбрать автоматически).

(*) Сделайте так, чтобы при открытии файла форма подстраивалась под размер изображения (если она не развернута на весь экран).

Дополнительную информацию можно найти в руководстве MSDN: <https://msdn.microsoft.com/ru-ru/library/dd492135.aspx>

Задача 2. Кассовый чек v.0.4.1

Реализуйте графический пользовательский интерфейс (GUI) для кассового чека. Создайте новый проект Windows Forms и перенесите в него файлы классов. Добавьте элементы управления для ввода-вывода данных. Подключите свойства и методы классов для ввода-вывода данных.

Контрольные вопросы

1. Как создать оконное приложение WindowsForms?
2. Чем является форма по сути?
3. Что такое событие?
4. Как создать обработчик события? Как удалить ненужный обработчик?
5. Что такое элементы управления?
6. Какие элементы управления вы знаете, для чего они предназначены?
7. За что отвечает свойство Dock элементов управления?
8. Как вывести окно с сообщением для пользователя?
9. Какие диалоговые стандартные диалоговые окна присутствуют в VisualStudio?
10. Как осуществляется чтение и запись в файл?
11. Что такое исключения?
12. Как применяется конструкция try-finally-catch?

ЗАКЛЮЧЕНИЕ

В пособии рассмотрены пять тем, охватывающих основные подходы к программированию на языке C# в среде VisualStudio. Каждая тема содержит краткое изложение теоретических положений и синтаксических конструкций C#, несколько примеров, в которых демонстрируются применение этих положений на практике и приемы разработки в среде VisualStudio, задачи для самостоятельного решения и контрольные вопросы.

В первой теме рассматриваются основы синтаксиса C# и создание консольных приложений с текстовым графическим интерфейсом: ввод-вывод данных, объявление переменных, арифметические операции, конкатенация строк. Демонстрируется разница между различными типами данных, приведение типов, различные операторы ветвления (if, тернарный оператор ?:, switch). Показана простейшая методика тестирования работы приложения и составления тестовых примеров.

Вторая тема посвящена изучению циклических операторов и связанных типов данных - строк и массивов. Сравняются три типа циклов: цикл-счетчик, цикл с предусловием и с постусловием. Рассматриваются основные встроенные методы работы со строками. Демонстрируется создание, заполнение массивов, методы класса Array.

В третьей теме рассмотрены основные понятия объектно-ориентированного программирования (ООП): класс, объект, атрибут, метод. На двух примерах показывается создание классов в программе, использование схемы классов и документирующих комментариев. Объясняется различие между хранимыми и ссылочными типами данных. Вводится понятие коллекции и списка, указываются их отличия от массивов.

Четвертая тема продолжает примеры третьей темы и демонстрирует применение принципов ООП на практике. Рассматривается применение свойств вместо полей и методов без параметров, области видимости членов класса, наследование, абстрактные классы, перегрузка и переопределение методов. Одновременно в примере демонстрируется создание текстового интерфейса с управлением через нажатие клавиш клавиатуры.

Пятая тема посвящена созданию графического пользовательского интерфейса. Также в примере рассматриваются понятие события, работа с файлами, вывод окон сообщений, обработка исключений, декомпозиция программы путем выделения методов и свойств.

Изложенный материал направлен на формирование базовых знаний по разработке приложений. Для дальнейшего развития в данном направлении требуется изучение современных практик программирования. Рекомендуется изучение следующих тем: выбросы и обработка исключений, коллекции (словари, деревья и др.), XML-сериализация, делегаты, лямбда-выражения, интерфейсы, модульное тестирования, совместная (командная) разработка.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

Albahari, J. *C# 7.0 in a Nutshell* / J. Albahari, B. Albahari. - O'Reilly Media, 2017. - 1088 p.

Вагнер, Б. Наиболее эффективное программирование на C#. 50 способов улучшения кода / Б. Вагнер. - Москва : Вильямс, 2017. - 240 с.

Видеокурс по C# Базовому. - URL: <https://www.youtube.com/playlist?list=PLvItDmb0sZw-kmcZAZJ29eTtAV56D5dgW>.

Документирование кода с помощью XML-комментариев. - URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/codedoc>.

Рекомендации по написанию кода на C#. - URL: <https://csharpcodingguidelines.com/> (перевод на русский язык <https://habr.com/post/272053/>).

Рихтер, Дж. *CLR via C#*. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# / Дж. Рихтер. - Санкт-Петербург : Питер, 2017. - 896 с.

Руководство по программированию на C#. - URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/index>.

Skeet, J. *C# in Depth, Third Edition* / J. Skeet. - Manning, 2013. - 616 p.

Соглашения о написании кода на C#. - URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>.

Троелсен, Э. *Язык программирования C# 7 для платформы .NET и .NET Core* / Э. Троелсен, Ф. Джепикс. - 8-е изд. - Москва : Диалектика, 2018. - 1328 с.

Учебное издание

Коробецкая Анастасия Александровна

**РАЗРАБОТКА
ПРОГРАММНЫХ ПРИЛОЖЕНИЙ**

Учебное пособие

Руководитель издательской группы О.В. Егорова
Редактор Т.В. Федулова
Корректор Л.И. Трофимова
Компьютерная верстка А.С. Емилиной

Подписано к изданию 26.12.2019. Печ. л. 12,88.
ФГБОУ ВО "Самарский государственный экономический университет".
443090, Самара, ул. Советской Армии, 141.