

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

УТВЕРЖДАЮ

Директор ИнЭО

_____ С.И. Качин

«_____» _____ 2016 г.

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Методические указания к выполнению лабораторных работ
для студентов ИнЭО, обучающихся по направлению
09.03.01 «Информатика и вычислительная техника»

Составитель И.П. Скирневский

Издательство
Томского политехнического университета
2016

УДК 004.42

Технологии программирования: метод. указ. к выполнению лабораторных работ для студентов ИнЭО, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» / сост. И.П. Скирневский; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2016. – 81 с.

Методические указания рассмотрены и рекомендованы к изданию методическим семинаром кафедры автоматики и компьютерных систем «____» _____ 2016 года, протокол № ____.

Зав. кафедрой АиКС,
доцент, кандидат техн. наук _____ А.С. Фадеев

Аннотация

Методические указания по выполнению лабораторных работ по дисциплине «Технологии программирования» предназначены для студентов ИнЭО, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника». Данные лабораторные работы выполняются в шестом и седьмом семестрах.

Приведены подробные методические указания по выполнению лабораторных работ по дисциплине «Технологии программирования», а также требования к оформлению отчетов по лабораторным работам.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	5
1. УСТАНОВКА И НАСТРОЙКА СРЕДЫ MICROSOFT VISUAL STUDIO.....	6
1.1. Среда разработки Microsoft Visual Studio	6
1.1.1. Общая информация	6
1.1.2. Загрузка и установка	7
1.1.3. Первый запуск.....	9
1.1.4. Про Solution Explorer	13
1.2. Структура простой программы	13
1.2.1. Ключевые элементы приложения HelloWorldApplication	15
1.2.2. Директива Using.....	16
1.3. Запуск программы	19
1.4. Самоконтроль.....	22
2. ЛАБОРАТОРНАЯ РАБОТА № 1. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ...	23
2.1. Основы структурного программирования	23
2.1.1. Циклы.....	24
2.1.2. Ветвления / Условные конструкции	28
2.1.3. Тернарная операция.....	30
2.2. Реализация приложения «Записная книжка».....	31
2.2.1. Создание хранилища записей.....	31
2.2.2. Внедрение циклов и ветвлений	32
2.3. Запуск программы	37
2.4. Задание на лабораторную работу № 1	37
3. ЛАБОРАТОРНАЯ РАБОТА № 2. ФУНКЦИОНАЛЬНЫЙ ПОДХОД	40
3.1. Ключевые особенности функционального подхода	40
3.2. Инициализация функции Main.....	41
3.3. Возвращаемый тип	42
3.4. Имя функции	42
3.5. Параметры функции	43
3.6. Тело функции	44
3.7. Структуры.....	44
3.7.1. Конструктор структуры	45
3.8. Модернизация приложения «Записная книжка»	46
3.9. Задание на лабораторную работу № 2	48
4. ЛАБОРАТОРНАЯ РАБОТА № 3. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ.....	50
4.1. Ключевые особенности ООП	50
4.2. Первый класс.....	51

4.3. Ключевое слово new	56
4.4. Понятие конструкторов.....	57
4.4.1. Стандартный конструктор	58
4.4.2. Определение специальных конструкторов	59
4.5. Основные принципы объектно-ориентированного программирования	60
4.5.1. Роль инкапсуляции	61
4.5.2. Роль наследования	63
4.5.3. Роль полиморфизма	68
4.6. Заключение.....	76
4.7. Задание на лабораторную работу № 3	77
5. ЛАБОРАТОРНАЯ РАБОТА № 4. РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ И ТЕХНОЛОГИЯ СОМ	79
5.1. Задание на лабораторную работу № 4	79

ВВЕДЕНИЕ

В рамках дисциплины «Технологии программирования» студенты познакомятся с базовыми технологиями программирования, начиная с функционального или так называемого, процедурного программирования. Параллельно осваивая методологию разработки программного обеспечения – структурное программирование.

В процессе выполнения лабораторных работ студенты познакомятся с объектно-ориентированным подходом, как с одной из наиболее удобных методологией программирования при разработке сложных программных продуктов, в основе которой лежит представления программы в виде связанных между собой объектов. Параллельно с изучением ООП студенты получают практические навыки реализации модульных приложений. Стоит отметить, что модульность программ в данном курсе будет реализована в самом примитивном понимании и такие фреймворки как Managed Extensibility Framework (MEF) затрагиваться не будут. Завершающей частью курса является использование технологии COM (Component Object Model – модель многокомпонентных объектов), предложенной компанией Microsoft.

В рамках курса студенты пройдут путь от создания небольших консольных приложений до создания оконных приложений с гибким графическим интерфейсом на базе самых популярных средств разработки и технологий.

1. УСТАНОВКА И НАСТРОЙКА СРЕДЫ MICROSOFT VISUAL STUDIO

1.1. Среда разработки Microsoft Visual Studio

1.1.1. Общая информация

Все лабораторные работы выполняются в среде Microsoft Visual Studio 2015 Community Edition. Visual Studio является одной из самых популярных сред разработки (IDE – Integrated development environment). На текущий момент Studio включает в себя все необходимое компоненты по созданию приложений для мобильных устройств, настольных приложений, веб-приложений и облачных решений. Доступна возможность писать код для iOS, Android и Windows в одной интегрированной среде разработки. Visual Studio обладает удобной и функциональной средой IntelliSense и рядом других преимуществ с которыми мы познакомимся по мере прохождения курса.

На рис. 1 представлена инфо-графика по продукту Visual Studio 2013 предоставленная ресурсом tadviser.ru.

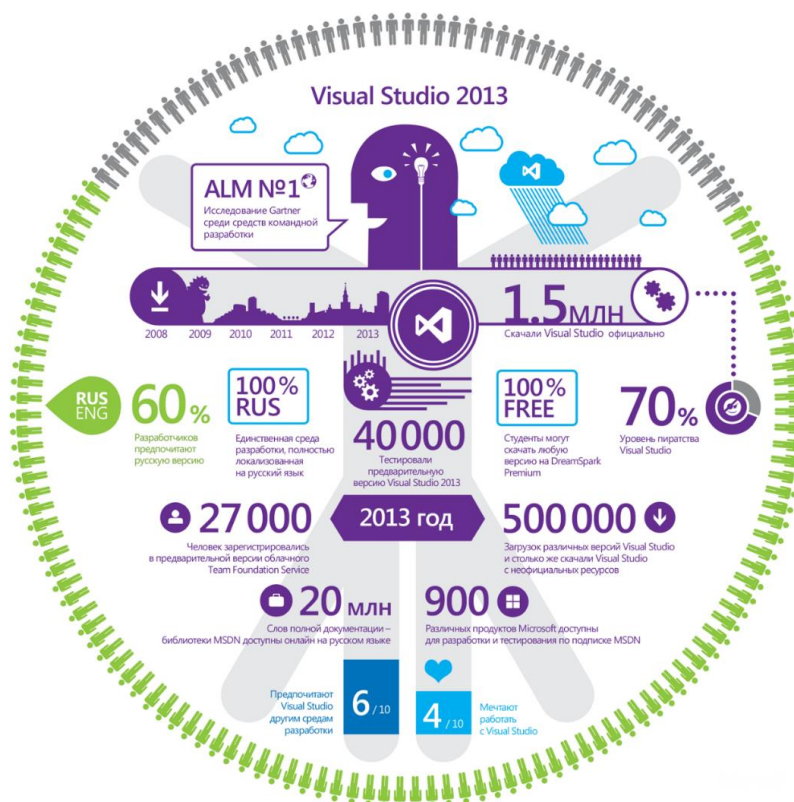


Рис. 1. Инфо-графика по продукту Visual Studio 2013

Продукт Visual Studio поставляется с конструкторами графических пользовательских интерфейсов, поддержкой фрагментов кода, инструментами манипулирования базами данных, утилитами для просмотра объектов и проектов, а также встроенной справочной системой. Visual Studio поддерживает множество дополнительных возможностей, наиболее важные из которых перечислены ниже:

- визуальные редакторы и конструкторы XML;
- поддержка разработки приложений для мобильных устройств Windows;
- поддержка разработки приложений для Microsoft Office;
- поддержка визуального конструктора для проектов Windows Workflow Foundation;
- встроенная поддержка рефакторинга кода;
- инструменты визуального конструирования классов.

С большинством перечисленных возможностей студенты познакомятся в процессе выполнения лабораторного практикума.

1.1.2. Загрузка и установка

Чтобы скачать Microsoft Visual Studio 2015 Community Edition перейдите по ссылке:

<https://www.visualstudio.com/downloads/download-visual-studio-vs>

и нажмите «Скачайте Community бесплатно» (рис. 2).

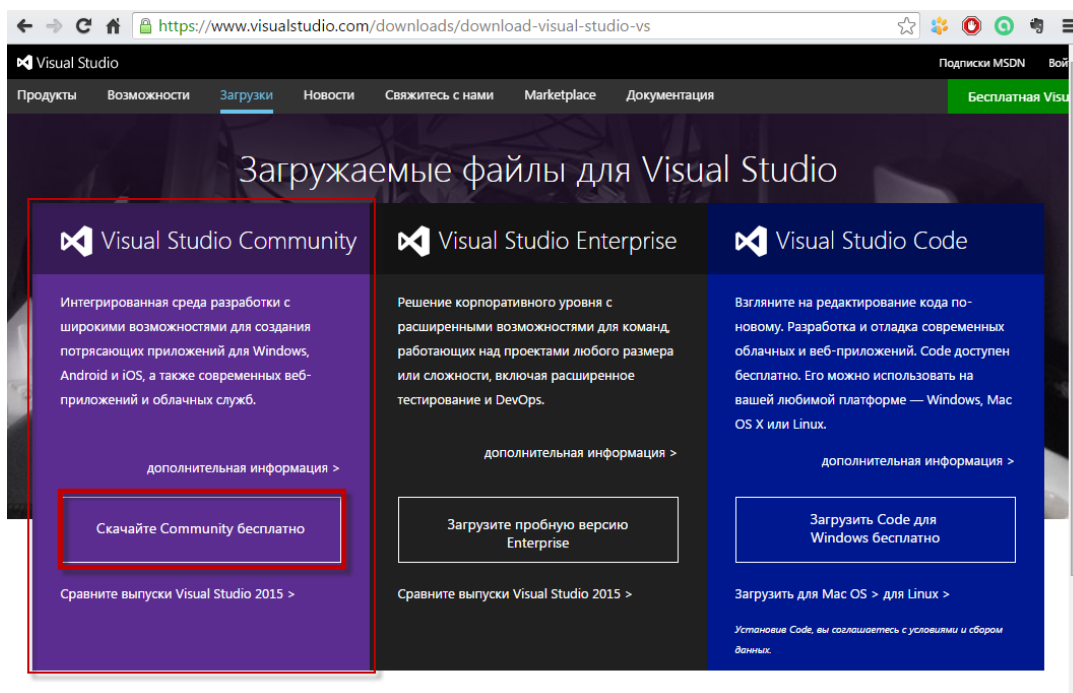


Рис. 2. Загрузка Visual Studio

Сохраните файл «vs_community_RUS.exe» в удобную для вас директорию на компьютере. Обратите внимание, что файл является веб-установщиком и все основные пакеты будут загружены на ваш компьютер во время процесса установки. Если вы планируете выполнить установку позже, перейти в раздел «Загружаемые файлы для Visual Studio» – «Visual Studio 2015» – «Community 2015» и выберете формат загружаемого файла ISO (рис. 3).

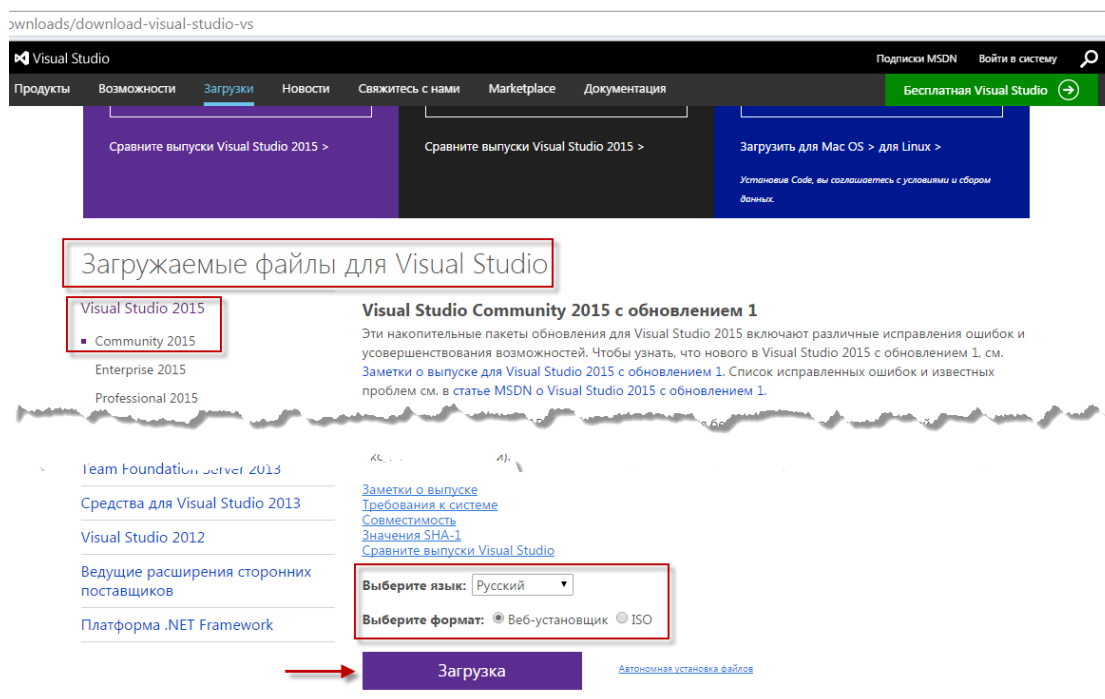


Рис. 3. Выбор формата

После запуска установочного файла выберите тип установки Typical и нажмите кнопку «Instal» (в русской версии продукта названия кнопок могут отличаться), рис. 4.

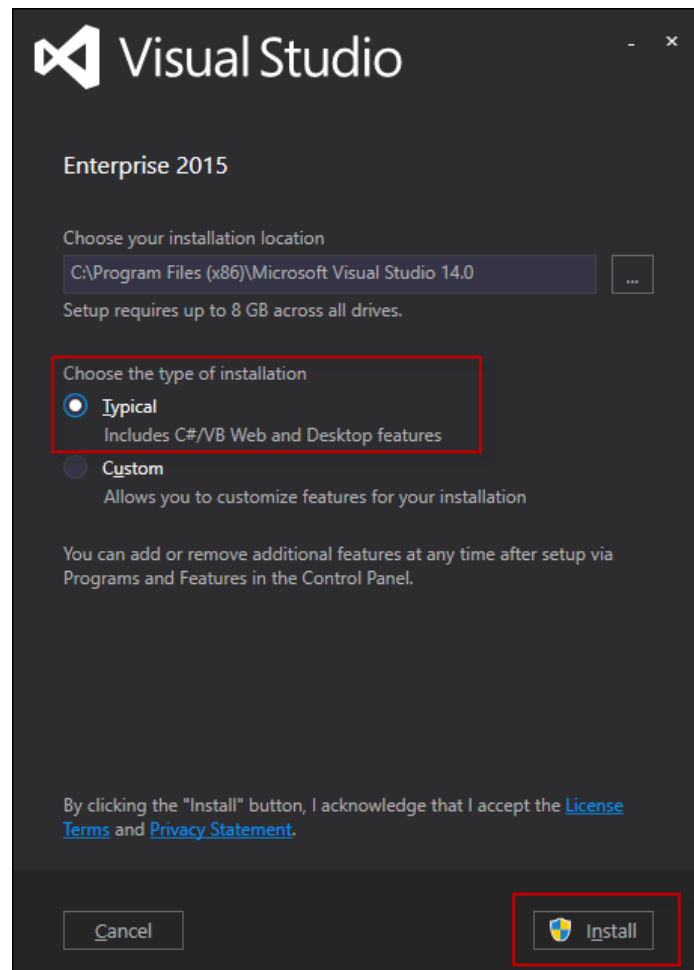


Рис. 4. Установка Visual Studio

1.1.3. Первый запуск

После успешной установки запустите Visual Studio из выбранной вами папки либо из меню Пуск. После запуска откроется экран приветствия, представленный на рис. 5, разберем его ключевые области.

1. Область вкладок. В данной области будут размещаться все открытые окна Visual Studio.

2. Область создания/открытия проекта.

3. Область последних открытых проектов.

В центральной части окна обычно размещены новости и другая вспомогательная информация. Стоит обратить внимание, что при использовании русской версии Visual Studio названия элементов будут отличаться. В правой части обычно размещается так называемый Обзорщик решения (Solution Explorer), представляющей из себя дерево всех файлов в рамках одного Решения (Solution) или проекта. Мы вернемся к изучению проводника по ходу выполнения лабораторных работ.

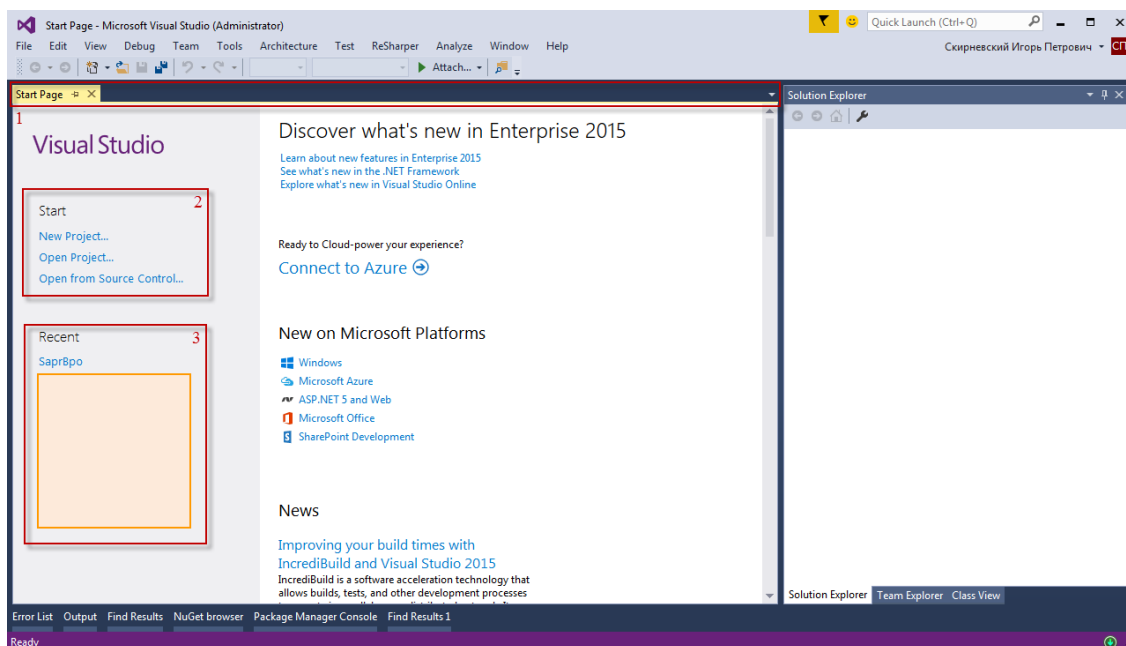


Рис. 5. Начальная страница

Для создания первого проекта нажмите на «Новый проект» (New Project...) в левой части стартового окна. Либо перейдите в меню File – New – Project (рис. 6, 7).

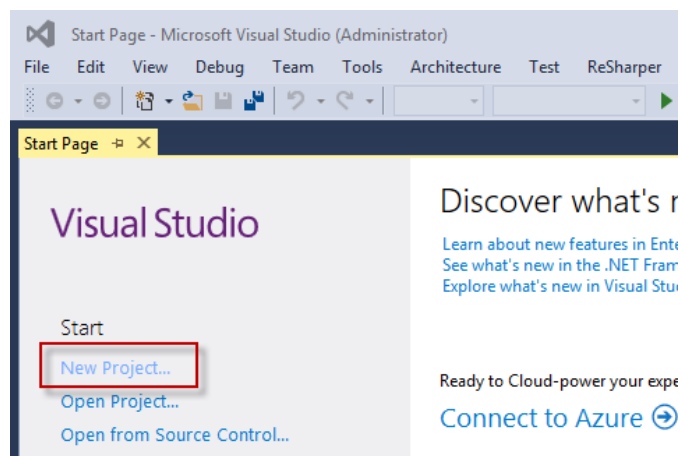


Рис. 6. Создание нового файла

Также создание нового проекта доступно через сочетание клавиш <Ctrl> + <Shift> + <N>. С подробным описанием всех «горячих клавиш» можно познакомиться, перейдя по ссылке <http://visualstudioshortcuts.com/2015>.

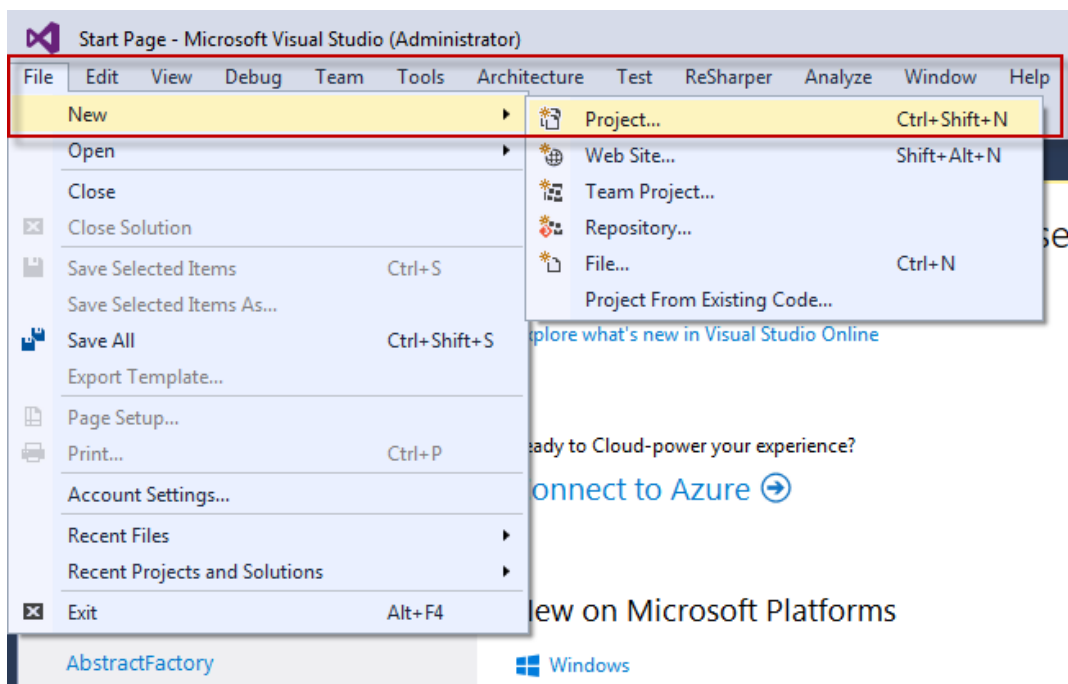


Рис. 7. Создание нового проекта через меню File

После нажатия на кнопку создания нового проекта, необходимо выбрать тип проекта. Для этого следует выбрать из раздела шаблонов (Templates) проект Visual C#. Все проекты, созданные в рамках нашего курса, будут выбраны из раздела Visual C#. Стоит также обратить внимание, что в среде Visual Studio существует множество других типов проектов: Visual Basic, F#, C++ и другие.

Для выполнения первой лабораторной работы необходимо выбрать проект Visual C# – Console Application. Обращаю ваше внимание на то, что шаблон проекта Console Application есть в различных подразделах меню. Чтобы не ошибиться перейдите в раздел Windows, как показано на рис. 8.

В разделе выбора Framework можно ничего не менять и оставить предложенную средой версию. **Важно** корректно заполнить поля Name, Location и Solution Name, которые находятся в нижней части экрана.

Поле Name – Название проекта. При заполнении поля Name, поле Solution заполняется тем же именем автоматически. Студентам не следует создавать отдельную директорию под свой проект, Visual Studio создаст ее автоматически.

Пример: Если пользователь создает проект с именем TestProject в директории VS/Projects/, то папка TestProject будет создана автоматически. Если предварительно создать директорию под проект получится следующий не самый удобный путь:

VS/Projects/ TestProjec/TestProject/

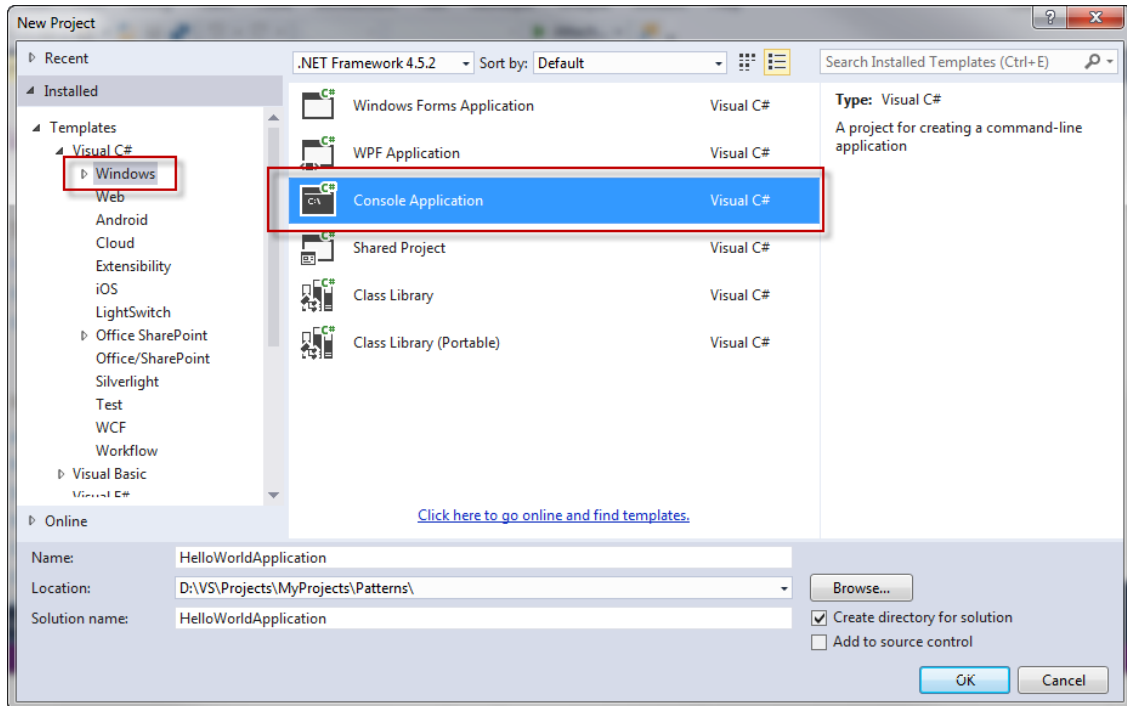


Рис. 8. Создание консольного приложения

После создания нового консольного приложения Visual Studio открывает файл Program.cs по умолчанию и Solution Explorer (Обозреватель решения) в правой части окна (рис. 9).

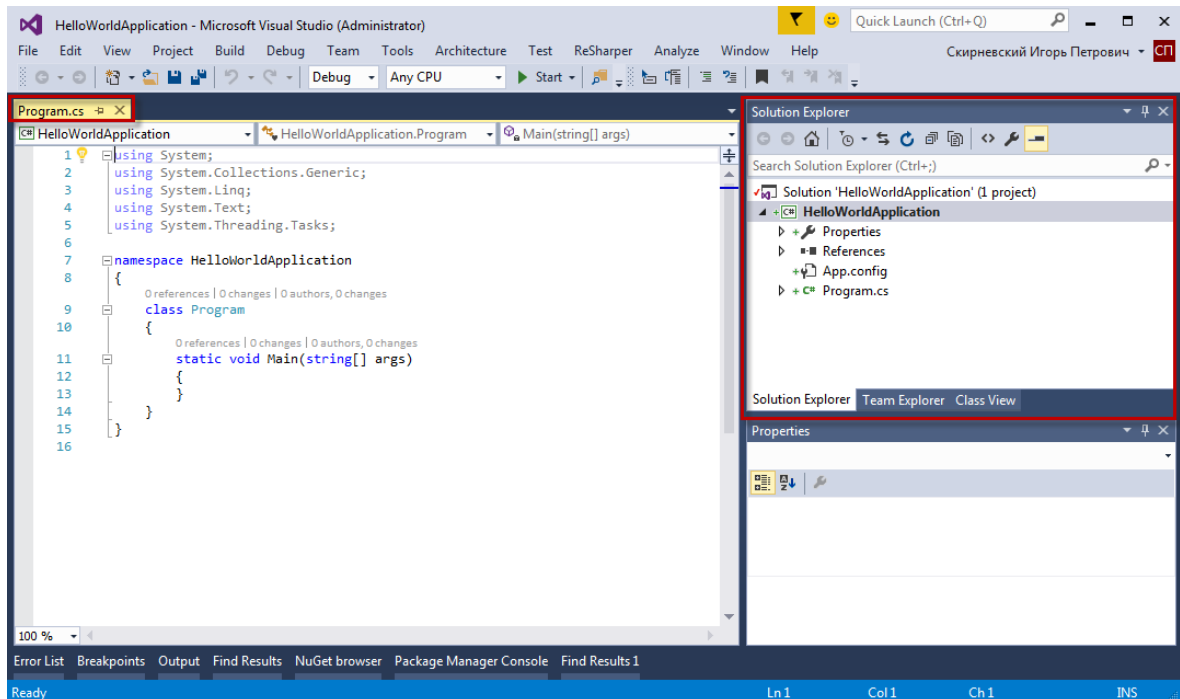


Рис. 9. Файл Program.cs после создания нового проекта

1.1.4. Про Solution Explorer

Утилита Solution Explorer (Проводник решений), доступная через меню View (Вид), позволяет просматривать набор всех файлов содержимого и ссылочных сборок, которые входят в состав текущего проекта (рис. 9). Также обратите внимание, что заданный файл (например, Program.cs) можно раскрыть, чтобы просмотреть определенные в нем кодовые типы. По ходу изложения материала будут указываться и другие полезные особенности Solution Explorer. Однако при желании можете самостоятельно поэкспериментировать с каждой из опций. Кроме того, внутри папки References (Ссылки) в окне Solution Explorer отображается список всех сборок, на которые имеются ссылки. В зависимости от типа выбираемого проекта и целевой версии .NET Framework, этот список выглядит по-разному. Поскольку мы создали консольное приложение, набор автоматически включаемых библиотек минимален (System.dll, System.Core.dll, System.Data.dll и т.д.).

1.2. Структура простой программы

Язык C# требует, чтобы вся логика программы содержалась внутри определения типа. В отличие от многих других языков, в C# не допускается создание ни глобальных функций, ни глобальных элементов данных. Вместо этого все данные-члены и все методы должны содержаться внутри определения типа. Говоря простым языком, весь код, который будет создан в рамках любого проекта, будет написан внутри классов или структур, фактически не существует кода «висящего» в пространстве. Мы уже создали первый проект HelloWorldApplication и, как показано ниже, в исходном коде Program.cs нет ничего особо примечательного, тем ни менее давайте разберем структуру программы:

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

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

С учетом приведенного кода, модифицируем метод `Main ()` класса `Program`, добавив в него следующие операторы:

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

namespace HelloWorldApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Вывести пользователю простое сообщение.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();
            // Ожидать нажатия клавиши <Enter> перед завершением работы.
            Console.ReadLine();
        }
    }
}
```

На заметку! C# является языком программирования, чувствительным к регистру. Следовательно, `Main` – не то же самое, что `main`, а `Readline` – не то же самое, что `readLine`. Запомните, что все ключевые слова C# вводятся в нижнем регистре (например, `public`, `lock`, `class`, `dynamic`), а названия пространств имен, типов и членов начинаются (по соглашению) с заглавной буквы и содержат заглавные буквы в любых вложенных в них словах (как, например, `Console.WriteLine`, `System.Windows.MessageBox` и `System.Data.SqlClient`). Как правило, при каждом получении от компилятора ошибки, связанной с неопределенными символами, в первую очередь следует проверить регистр символов и точность написания имен.

И так, мы имеем определение типа класса, который поддерживает единственный метод по имени `Main()`. По умолчанию среда Visual Studio назначает классу, определяющему метод `Main()`, имя `Program`, однако, при желании это можно изменить. Каждое исполняемое приложение C# (консольная программа, программа рабочего стола Windows или Windows-служба) должно содержать класс, определяющий метод `Main()`, который используется для обозначения точки входа в приложение.

Формально класс, который определяет метод `Main()`, называется объектом приложения. Хотя в одном исполняемом приложении разре-

шено иметь несколько объектов приложений (это может быть удобно при модульном тестировании), потребуется проинформировать компилятор о том, какой из методов `Main()` должен использоваться в качестве точки входа.

Обратите внимание, что сигнатура метода `Main()` снабжена ключевым словом `static`, о котором будет написано ниже. Пока же достаточно знать, что область действия статических членов охватывает уровень класса (а не уровень объектов), поэтому они могут вызываться без предварительного создания нового экземпляра класса.

За время прочтения абзацев, написанных выше, студенты познакомились с рядом не знакомых слов и, если читателю начинает казаться, что он запутался и уже ничего не понимает, отчаиваться не стоит, в конце данного раздела мы разберем все новые слова подробно.

Кроме ключевого слова `static` этот метод `Main ()` принимает один параметр, который представляет собой массив строк (`string [] args`). Хотя в текущий момент этот массив никак не обрабатывается, данный параметр может содержать любое количество входных аргументов командной строки. И, наконец, этот метод `Main ()` был сконфигурирован с возвращаемым значением `void`, которое означает, что мы не определяем явно возвращаемое значение с помощью ключевого слова `return` перед выходом из области действия метода.

Логика `Program` содержится внутри самого метода `Main ()`. Здесь используется класс `Console`, который определен в пространстве имен `System`. В состав его членов входит статический метод `WriteLine ()`, который позволяет отправлять строку текста и символ возврата каретки на стандартное устройство вывода. Кроме того, здесь вызывается метод `Console.ReadLine ()`, чтобы окно командной строки, запускаемое в IDE-среде `Visual Studio`, оставалось видимым во время сеанса отладки до тех пор, пока не будет нажата клавиша `<Enter>`.

1.2.1. Ключевые элементы приложения `HelloWorldApplication`

Давайте рассмотрим структуру первого приложения еще раз, для удобства читателя, код, который приведен выше, скопирован еще раз в данный раздел.

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

namespace HelloWorldApplication
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            // Вывести пользователю простое сообщение.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();
            // Ожидать нажатия клавиши <Enter> перед завершением работы.
            Console.ReadLine();
        }
    }
}

```

1.2.2. Директива Using

В C# ключевое слово `using` упрощает процесс добавления ссылок на типы, определенные в заданном пространстве имен. Другими словами, `using` подключает необходимые библиотеки к вашему проекту. Так, если вы решили в коде своего приложения использовать функции работы с файлами вам следует добавить в проект пространство имен `System.IO` (приведено только в качестве примера). Возникает логичный вопрос, как держать в голове, все нужные подключения и где получить информацию по нужным библиотекам?

На самом деле, среда Visual Studio сама подскажет, какого пространства имен не хватает для работы и предложит добавить его в код проекта (рис. 10).

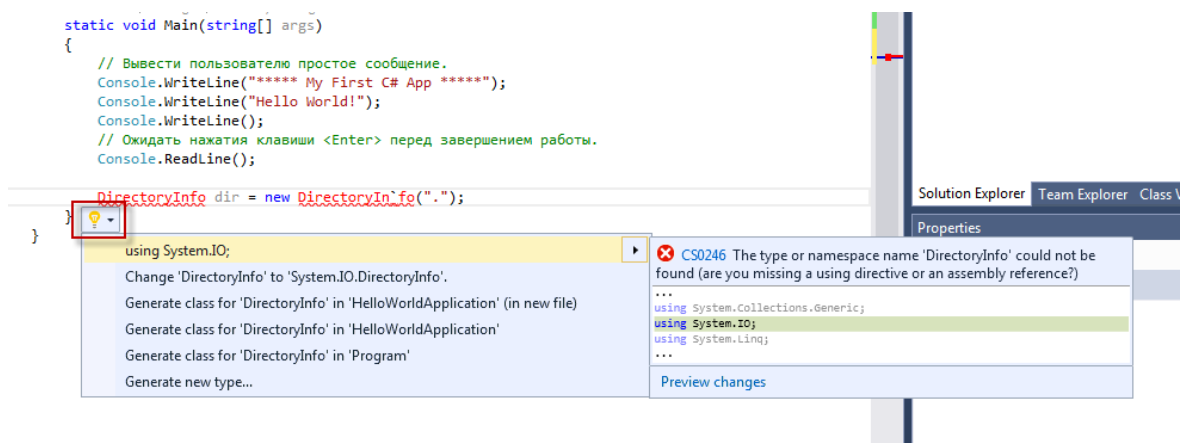


Рис. 10. Подсказка о пропущенном пространстве имен

Также всегда можно воспользоваться справкой, для этого выделите участок кода и нажмите F1.

Ниже приведена табл. 1 с основными типами, которые могут понадобиться студентам в своих проектах. Но, как можно было заметить, Visual Studio при создании пустого проекта самостоятельно добавляет ряд подключений самых популярных библиотек, что значительно упрощает разработку.

Таблица 1

Пространства имен .NET

Пространство имен .NET	Описание
System	Внутри пространства имен System содержится множество полезных типов, предназначенных для работы с внутренними данными, математическими вычислениями, генерацией случайных чисел, переменными среды и сборкой мусора, а также ряд часто применяемых исключений и атрибутов
System.Collections System.Collections.Generic	Эти пространства имен определяют набор контейнерных типов, а также базовые типы и интерфейсы, которые позволяют строить настраиваемые коллекции
System.Data System.Data.Common System.Data.EntityClient System.Data.SqlClient	Эти пространства имен используются для взаимодействия с базами данных через ADO.NET
System.IO System.IO.Compression System.IO.Ports	Эти пространства имен определяют множество типов, предназначенных для работы с файловым вводом-выводом, сжатием данных и портами
System.Drawing System.Windows.Forms	Эти пространства имен определяют типы, применяемые для построения настольных приложений с использованием исходного инструментального набора .NET для создания пользовательских интерфейсов (Windows Forms)
System.Windows System.Windows.Controls System.Windows.Shapes	Пространство имен System.Windows является корневым для нескольких пространств имен, которые представляют инструментальный набор для построения пользовательских интерфейсов Windows Presentation Foundation (WPF)
System.Linq System.Xml.Linq System.Data.DataSetExtensions	Эти пространства имен определяют типы, применяемые во время программирования с использованием API-интерфейса LINQ
System.Xml	В пространствах имен, связанных с XML, содержатся многочисленные типы, используемые для взаимодействия с XML-данными

1. Namespace (пространство имен)

Мы не будем подробно останавливаться на следующем участке кода, а именно пространстве имен. Заметим лишь, что при создании нового проекта, в нашем случае, HelloWorldApplication Visual Studio самостоятельно создает одноименное пространство имен.

Объявление собственного пространства имен поможет в управлении областью действия имен классов и методов в крупных программных проектах.

2. Class Program

В первой лабораторной работе не будет затрагиваться тема объектно-ориентированного программирования, по этому строчку `class Program` мы пропустим и вернемся к ней в следующих лабораторных работах. Единственное, что стоит отметить, как писалось выше, весь код, написанный в рамках платформы .NET должен принадлежать какому-либо классу или структуре. В нашем случае, даже самая примитивная программа находится в рамках класса `Program` и размещается в нем автоматически.

3. Static void Main(string[] args)

Данная строка описывает функцию `Main` – входную точку программы. Функции будут детально рассмотрены в следующей лабораторной работе.

Внутри фигурных скобок функции `Main` расположен основной код программы, структура которого в данной лабораторной выводит строчки `***** My First C# App *****` и `"Hello World! "` на экран.

Вывод на консоль осуществляется при помощи статического метода `WriteLine` класса `Console`, который принимает на вход строку. Под словом «метод» в контексте данной лабораторной работы можно понимать некоторую функцию. Как вы думаете, функция `WriteLine` имеет возвращаемый тип? Или является `void` (т.е. ничего не возвращает)?

Строка `Console.ReadLine()` необходима для того, чтобы консоль не закрылась автоматически после вывода сообщений на экран, а дождалась нажатия клавиши пользователем. Попробуйте закомментировать данную строку и запустите приложение еще раз (запуск проекта подробно описан в следующей части).

Справка!

*Закомментированный код игнорируется компилятором и нужен для создания примечаний и пояснений к коду. В языке C# существует два вида комментариев. Однострочный: `///
»` два следа – комментирует только одну строчку. И многострочный: `/* текст */`.*

1.3. Запуск программы

После того, как код написан, необходимо выполнить запуск и отладку приложения. Для того, чтобы запустить приложение, необходимо нажать <F5> или воспользоваться кнопкой «Start» в среде Visual Studio (рис. 11).

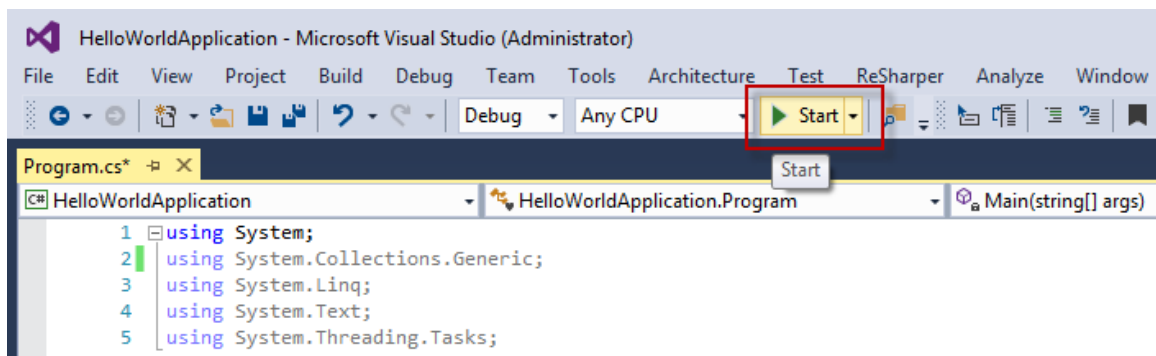


Рис. 11. Запуск проекта

Расширенный список опций доступен в разделе меню Debug (рис. 12).

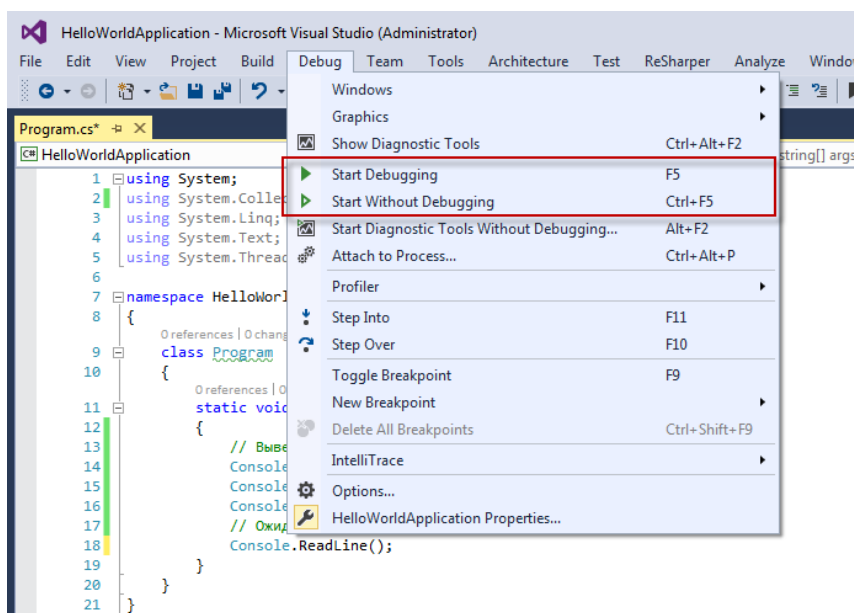


Рис. 12. Меню Debug

Если никаких ошибок не было найдено, приложение будет успешно запущено, и откроется консоль с текстом (рис. 13).

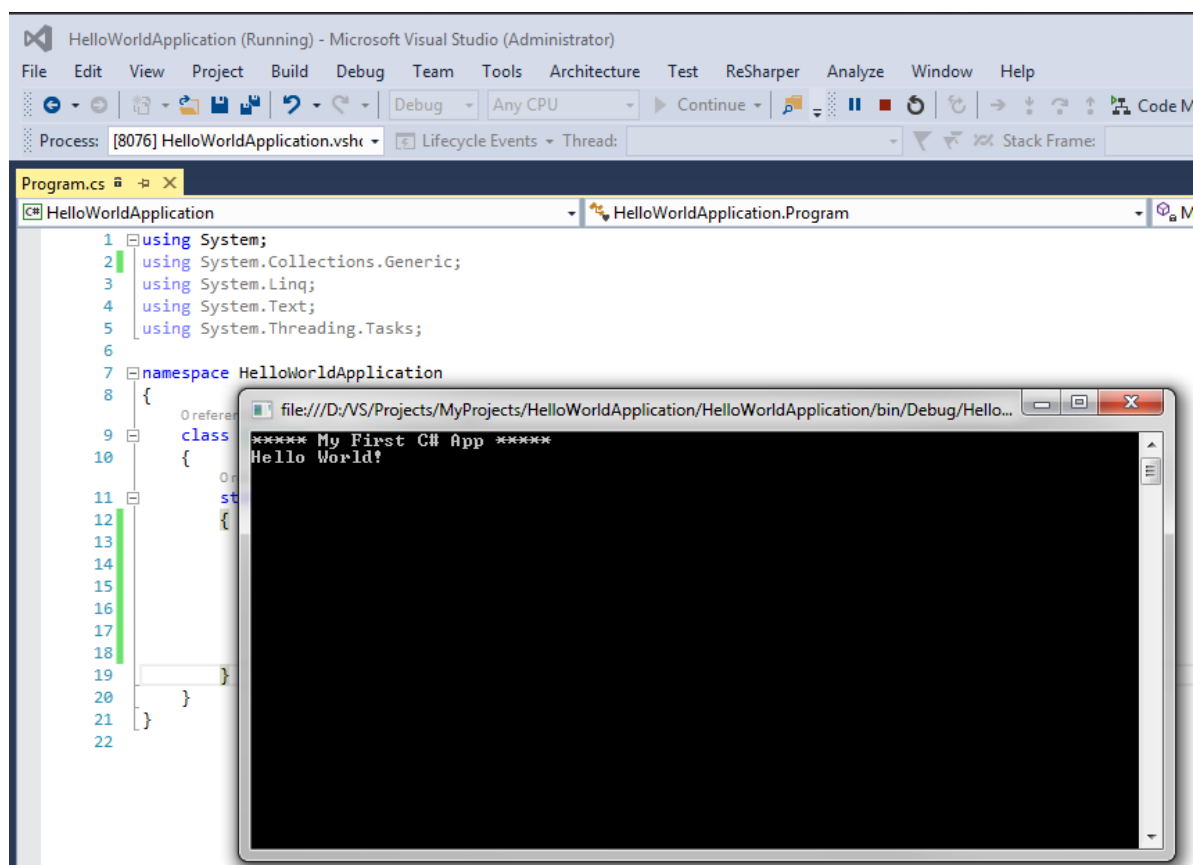


Рис. 13. Запуск проекта

Для завершения работы приложения можно нажать любую клавишу, либо закрыть окно. Также остановить запущенную программу позволяет сочетание клавиш **<Shift> + <F5>**.

Если в процессе анализа исходного кода компилятор обнаружит синтаксические ошибки (ошибки компиляции), Visual Studio сообщит о месте локализации ошибки.

Добавим в проект строку, как показано на рис. 14, и попробуем запустить программу.

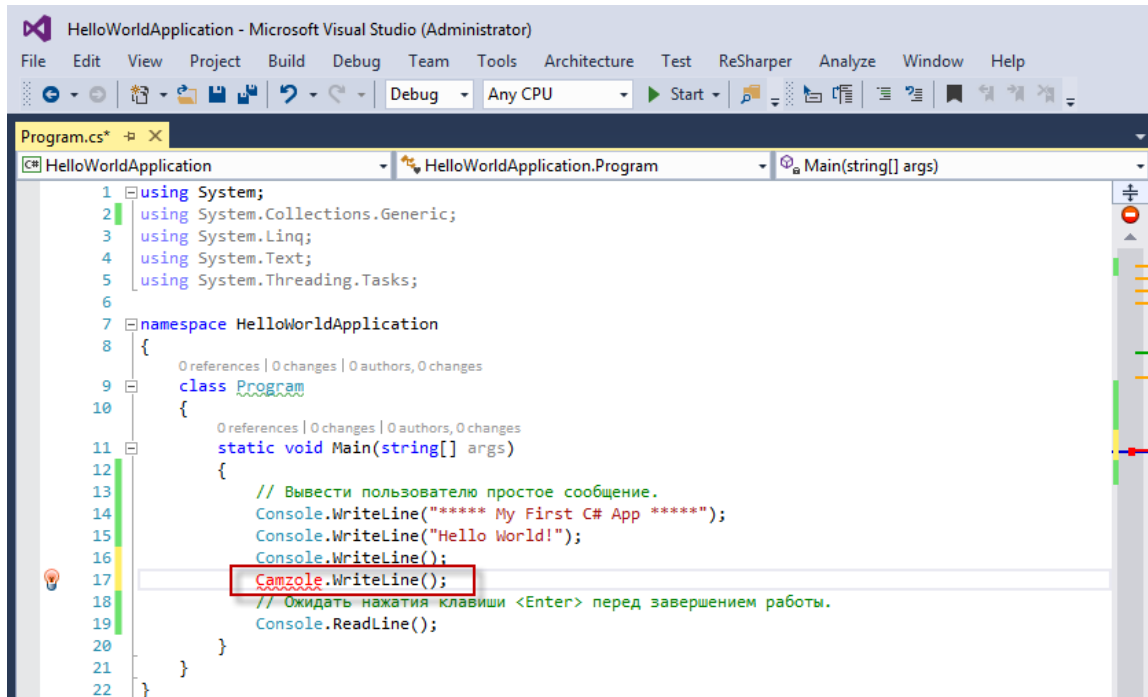


Рис. 14. Синтаксическая ошибка в коде проекта

После запуска приложения (клавиша <F5>, для тех, кто забыл) компилятор сообщит об ошибке и приложения не будет запущено (рис. 15).

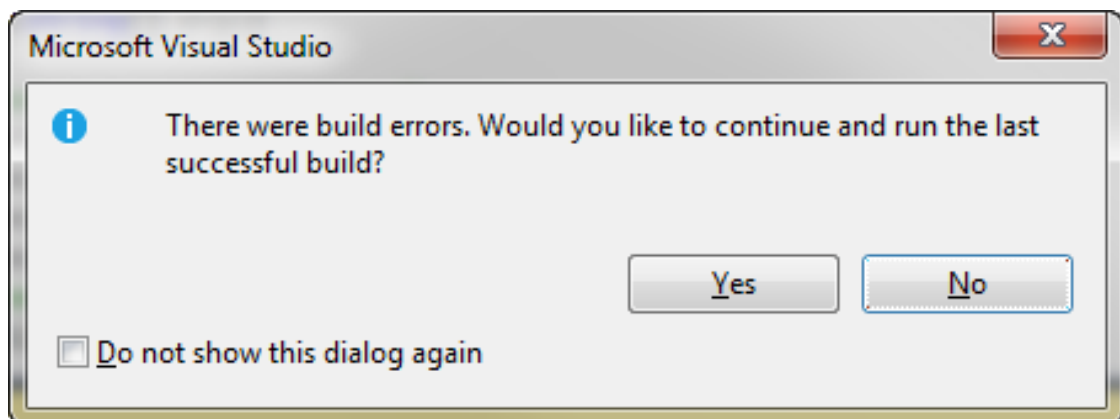


Рис. 15. Предупреждение о наличии ошибок

После нажатия на кнопку «No», автоматически всплывет окно Error List с детальным отчетом об ошибке (рис. 16).

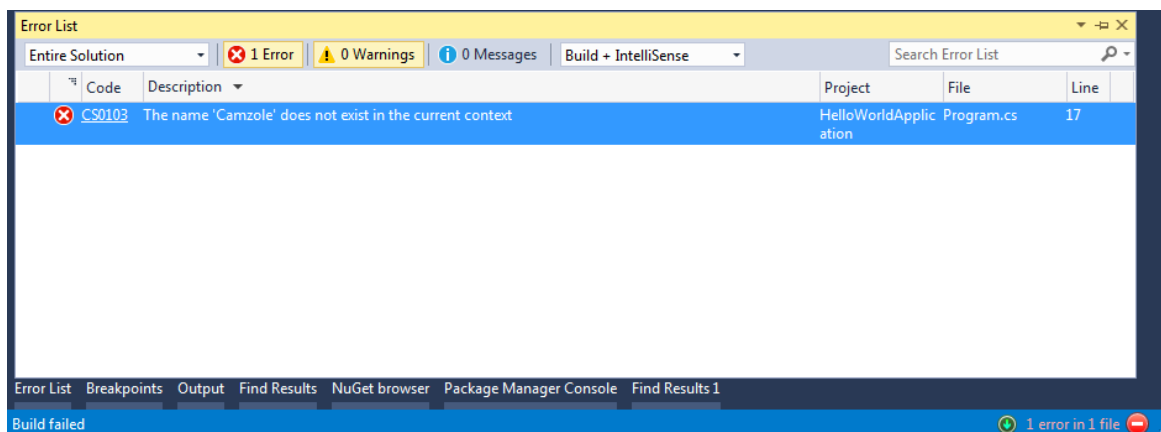


Рис. 16. Сообщение об ошибке

Таблица состоит из нескольких столбцов. Наиболее интересные из которых «Description» и «Line». В нашем случае ошибка звучит как: «*The name 'Camzole' does not exist in the current context*», что можно трактовать как, не существует имени Camzole в текущем контексте. Это значит, что в нашем проекте нет функции или класса с таким именем, ровно, как и в подключенных нами ранее библиотеках.

Колонка Line указывает номер строки, где произошла ошибка, в нашем случае это строка 17. Двойное нажатие на строку ошибки перебросил курсор на строчку в коде проекта.

Другие тонкости отладки будут рассмотрены в следующих лабораторных работах. На этом выполнение первой лабораторной закончено.

1.4. Самоконтроль

1. Установить среду Microsoft Visual Studio
2. Создать проект «Консольное приложение C#». Для названия проекта используйте следующий синтаксис Lab#_Familia. Например, Lab1_Ivanov. **Лабораторные работы с неправильно оформленным названием могут быть не приняты или случайно удалены.**
3. Вывести на экран две строки:
 - a. В первой строке «Hello World!»
 - b. Во второй строке: ФИО и номер зачетной книжки.
4. Удалить из проекта все ненужные блоки using (которые были добавлены средой по умолчанию).
5. Добавить в проект строку, содержащую ошибку, и попытаться запустить приложение.
6. Исправить ошибку и повторить запуск.
7. Оформить отчет с описанием хода выполнения вашей лабораторной работы.

2. ЛАБОРАТОРНАЯ РАБОТА № 1

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

2.1. Основы структурного программирования

Структурное программирование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Методология предложена в 1970-х годах Э. Дейкстрой и др.

В соответствии с данной методологией любая программа строится без использования оператора **goto** из трёх базовых управляющих структур: последовательность, ветвление, цикл; кроме того, используются подпрограммы, речь о которых пойдет в следующей лабораторной работе. При этом разработка программы ведётся пошагово, методом «сверху вниз». В теории программирования уже давно было доказано, что для решения задач совершенно любой сложности, начиная от примитивных окон и заканчивая сложными информационными системами можно составить программу, которая будет состоять только из трех структур, о которых было упомянуто выше. Официально такое исследование в 1966 году было проведено Боймом Якопини.

Схематично, структуры следование, цикл и ветвление можно представить следующим образом (рис. 17).

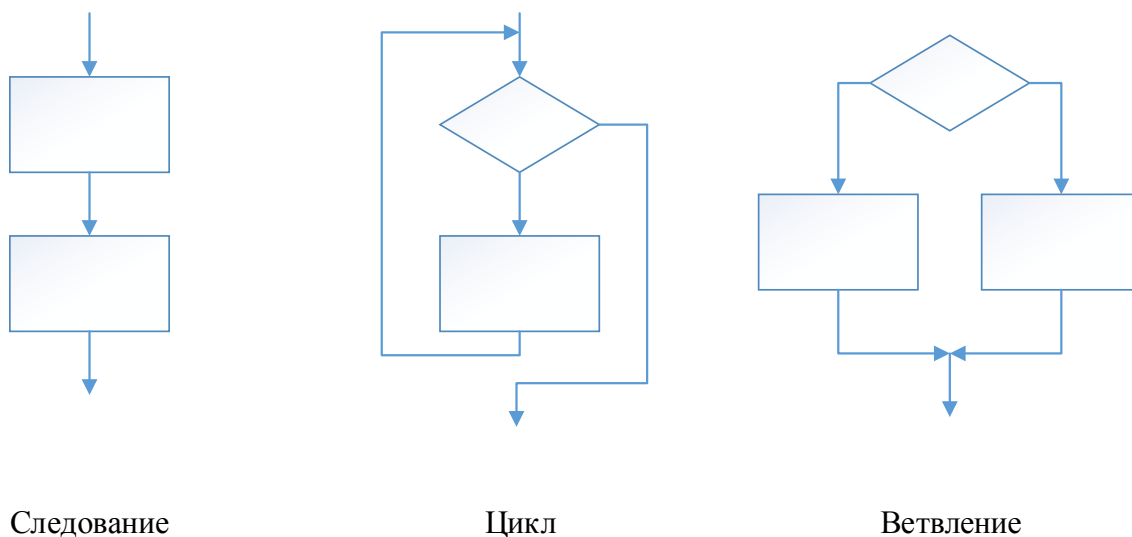


Рис. 17. Управляющие структуры

Ниже приведено краткое описание каждой структуры:

- **следованием** называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных);

- **цикл** задает многократное выполнение оператора;

- **ветвление** задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

Идеей использования базовых конструкций является получение программы простой структуры. Такую программу легко читать (а программы чаще приходится читать, чем писать), отлаживать и при необходимости вносить в нее изменения. Структурное программирование, как вы могли заметить, часто называли «программированием без **goto**», и в этом есть большая доля правды: частое использование операторов передачи управления в произвольные точки программы затрудняет прослеживание логики ее работы.

В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в C# есть четыре вида циклов и два вида ветвлений (на два и на произвольное количество направлений). Они введены для удобства программирования, и в каждом случае надо выбирать наиболее подходящие средства. Главное, о чем нужно помнить даже при написании самых простых программ, — что они должны состоять из четкой последовательности блоков строго определенной конфигурации.

2.1.1. Циклы

Как было указано выше, циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

1. **for**
2. **foreach**
3. **while**
4. **do...while**

Давайте разберем каждый вид отдельно и рассмотрим несколько примеров.

Цикл **for**

Цикл **for** имеет следующее формальное определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
```


Рассмотрим небольшой пример использования цикла `for`. Стандартный пример, прохождение массива из 10 элементов.

```
for (int i = 0; i < 9; i++)
{
    Console.WriteLine("Квадрат числа {0} равен {1}", i, i * i);
}
```

Первая часть объявления цикла — `int i = 0` — создает и инициализирует счетчик `i` (буква `i` выбрана в связи с устоявшейся традицией, хотя использоваться может и любой другой символ). Счетчик необязательно должен представлять тип `int`. Это может быть и другой числовой тип, например, `float`. И перед выполнением цикла его значение будет равно 0. В данном случае это тоже самое, что и объявление переменной.

Вторая часть — условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока `i` не достигнет 9.

И третья часть — приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: `i--`.

В итоге блок цикла сработает 9 раз, пока значение `i` не станет равным 9. И каждый раз это значение будет увеличиваться на 1. Нам необязательно указывать все условия при объявлении цикла. Например, цикл может быть реализован следующим образом:

```
int i = 0;
for (; ; )
{
    Console.WriteLine("Квадрат числа {0} равен {1}", ++i, i * i);
    System.Threading.Thread.Sleep(500);
}
```

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: `for (; ;)`. У нас нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно — бесконечный цикл.

Мы также можем опустить ряд блоков, как показано в примере ниже.

```
int i = 0;
for (; i < 9;)
{
    Console.WriteLine("Квадрат числа {0} равен {1}", ++i, i * i);
}
```

Этот пример по сути эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке `for`.

Цикл `foreach`

Исторически сложилось, что одним из наиболее трудных для понимания циклов является цикл – `foreach`. На самом деле в нем ничего сложно и сейчас мы в этом убедимся.

Цикл `foreach` предназначен для перебора элементов в контейнерах. Формальное объявление цикла `foreach`:

```
foreach (тип_данных название_переменной in контейнер)
{
    // действия
}
```

Для того чтобы ситуация стала более понятной, приведем небольшой пример. Допустим, у нас есть массив из 5 элементов и нам надо обратиться к каждому из элементов массива, чтобы вывести его значение на экран. Код приведен ниже.

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
foreach (int i in array)
{
    Console.WriteLine(i);
}
```

Обратите внимание, что в примере в качестве контейнера выступает массив данных типа `int`. Поэтому мы объявляем переменную с типом `int`.

Внимательный читатель может заметить, что абсолютно тоже можно сделать и при помощи цикла `for`. Вы можете проделать этот пример самостоятельно. Несмотря на лаконичную конструкция цикла `foreach`, цикл `for` более гибкий по сравнению с `foreach`. Если `foreach` последовательно извлекает элементы контейнера и только для чтения, то в цикле `for` мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы.

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < array.Length; i++)
{
    array[i] = array[i] * 2;
    Console.WriteLine(array[i]);
}
```

Разберите самостоятельно, что делает пример приведенный выше.

Цикл `do`

В цикле `do` сначала выполняется код цикла, а потом происходит проверка условия в инструкции `while`. И пока это условие истинно, цикл повторяется. Например, следующий код уменьшает значение переменной `i` на единицу, до тех пор, пока переменная больше нуля.

```
int i = 6;
do
{
    Console.WriteLine(i);
    i--;
}
while (i > 0);
```

Наверно, вы уже догадались, что код цикла сработает 6 раз, пока `i` не станет равным нулю. Но важно отметить, что цикл `do` гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции `while` не будет истинно. Фактически, задавая начальное значение для `i = -1` цикл гарантированно выполнится один раз. Попробуйте проверить это самостоятельно.

Цикл `while`

В отличие от цикла `do` цикл `while` сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int i = 6;
while (i > 0)
{
    Console.WriteLine(i);
    i--;
}
```

Попробуйте задать начальное условие для переменной `i` равное `-1` и запустить цикл. Сравните полученный результат с циклом `do...while`.

Операторы `continue` и `break`

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором `break`. Пример приведен ниже.

```
int[] array = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < array.Length; i++)
{
    if (array[i] > 10)
        break;
    Console.WriteLine(array[i]);
}
```

Поскольку в цикле идет проверка, больше ли элемент массива 10. То мы никогда не увидим на консоли последние два элемента, так как, увидев, что элемент массива больше 10, сработает оператор `break`, и цикл завершится.

Теперь поставим себе другую задачу. А что, если мы хотим, чтобы при проверке цикл не завершался, а просто переходил к следующему элементу. Для этого мы можем воспользоваться оператором `continue`:

```
int[] array = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < array.Length; i++)
{
    if (array[i] > 10)
        continue;
    Console.WriteLine(array[i]);
}
```

В этом случае цикл, когда дойдет до числа 12, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующему элементу массива.

2.1.2. Ветвления / Условные конструкции

Условные конструкции – один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей в зависимости от определенных условий.

В языке C# используются следующие условные конструкции: `if..else` и `switch..case`

Конструкция `if/else`

Конструкция `if/else` проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код. Давайте рассмотрим небольшой пример использования оператора.

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
    Console.WriteLine("Число {0} больше числа {1}", num1, num2);
}
```

После ключевого слова `if` ставится условие. И если это условие выполняется, то срабатывает код, который помещен далее в блоке `if` после фигурных скобок. В качестве условий выступают ранее рассмотренные операции сравнения.

В данном случае у нас первое число больше второго, поэтому выражение `num1 > num2` истинно и возвращает `true`, следовательно, управление переходит к строке `Console.WriteLine("Число {0} больше числа {1}", num1, num2);`

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок **else**:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
    Console.WriteLine("Число {0} больше числа {1}", num1, num2);
}
else
{
    Console.WriteLine("Число {0} меньше числа {1}", num1, num2);
}
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию **else if**, мы можем обрабатывать дополнительные условия:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
    Console.WriteLine("Число {0} больше числа {1}", num1, num2);
}
else if (num1 < num2)
{
    Console.WriteLine("Число {0} меньше числа {1}", num1, num2);
}
else
{
    Console.WriteLine("Число num1 равно числу num2");
}
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2 && num1 > 8)
{
    Console.WriteLine("Число {0} больше числа {1}", num1, num2);
}
```

Обратите внимание на конструкцию **&&**, которая выполняет логическое умножение, другими словами операцию «И». Это говорит о том, что блок будет истинен только при выполнении обоих условий. Не трудно догадаться, что существует логическое сложение, оператор «ИЛИ» – в языке C# обозначается как **<|>** две прямых черты.

Важно!

Распространенной ошибкой является использование оператора присваивания «=» внутри блока `if`. Важно понимать отличие между оператором присваивания и оператором проверки на равенство «==». Внутри блока `if` может использоваться только оператор проверки на равенство, так как он может вернуть значение истина или ложь, в зависимости от того, равны переменные или нет. Таким образом правильно будет `if (a == b)`. Будьте внимательны.

Конструкция switch

Конструкция `switch/case` аналогична конструкции `if/else`, так как позволяет обработать сразу несколько условий. Рассмотрим пример.

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
    case "Y":
        Console.WriteLine("Вы нажали букву Y");
        break;
    case "N":
        Console.WriteLine("Вы нажали букву N");
        break;
    default:
        Console.WriteLine("Вы нажали неизвестную букву");
        break;
}
```

После ключевого слова `switch` в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора `case`. И если совпадение будет найдено, то будет выполняться определенный блок `case`. В конце блока `case` ставится оператор `break`, чтобы избежать выполнения других блоков. Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок `default`, как в примере выше.

2.1.3. Тернарная операция

Тернарную операция имеет следующий синтаксис:

[первый операнд – условие] ? [второй операнд] : [третий операнд].

Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно `true`, то возвращается второй операнд; если условие равно `false`, то

третий. В некотором роде, тернарная операция представляет собой сокращенный синтаксис операции `if/else`, но все-таки имеет ряд отличий. Пример ниже:

```
int x = 3;
int y = 2;
Console.WriteLine("Нажмите + или -");
string selection = Console.ReadLine();

int z = selection == "+" ? (x + y) : (x - y);
Console.WriteLine(z);
```

Здесь результатом тернарной операции является переменная `z`. Если мы выше вводим "+", то `z` будет равно второму операнду – $(x+y)$. Иначе `z` будет равно третьему операнду.

2.2. Реализация приложения «Записная книжка»

В качестве примера рассмотрим создание простой записной книжки. Читатель может либо проделать всю последовательность шагов и выполнить тестовый проект, либо сразу выполнять задание в соответствии со своим вариантом.

Хотелось бы обратить внимание, что данный пример не является оптимальным решением задачи. И в ходе прохождения лабораторного практикума, мы будем модернизировать его, используя новые технологии и подходы. А пока не будем забегать вперед и создадим новое консольное приложение под названием Notebook.

Примечание. В данной лабораторной работе не будет описано создание проекта и работа со средой Visual Studio. Предполагается, что на момент выполнения лабораторной работы студент уже обладает обозначенными знаниями, которые давались в лабораторной работе № 1 данного лабораторного практикума.

В процессе выполнения лабораторной работы мы будем добавлять весь код в функцию `Main` без использования каких-либо дополнительных конструкций.

2.2.1. Создание хранилища записей

В нашем случае в качестве будет использоваться обыкновенная переменная – строковый массив. Это говорит о том, что все данные, записанные в записную книжку, будут удалены после каждого закрытия программы. Согласен с вами, не самая удобная записная книжка. Для хранения строковых данных использовать несколько видов контейне-

ров. Одним из самых простых решений – массив строк. Мы пропустим данный вариант, так как он не является ключевой частью лабораторной работы. Для удобства реализации в качестве хранилища записей будущей записной книжки используем так называемый список `List`. Детально ознакомиться со всеми особенностями списка `List` можно, вызвав справку. Пока же просто будем использовать список `List<T>`, не вдаваясь в подробности данной структуры. И так, создаем список записей, как показано в примере ниже.

```
List<String> notes = new List<string>();
```

В данном случае `<String>` указывает на то, что наш список хранит строки. Одна строка – одна запись в списке. Конструкция `new List<string>()` нужна для инициализации нового списка. И так, наш новый список называется `notes`, это объект типа `List<String>`. Списки очень удобны в плане добавления и удаления записей. Так, для добавления новой записи в наш список `notes` воспользуемся командой `Add`. Добавим несколько первоначальных значений, как показано в примере ниже.

Добавим пару начальных записей.

```
notes.Add("Первая запись");  
notes.Add("Вторая запись");
```

И так, мы создали наше, хоть и временное, но хранилище данных. Теперь после каждого включения программы в нем будет находиться две записи. Самое время внедрить основные элементы структурного программирования, циклы и ветвления.

2.2.2. Внедрение циклов и ветвлений

Вся логика нашего приложения будет заключаться в том, что пользователю предоставляется список команд:

- добавить запись;
- удалить запись по номеру n ;
- посмотреть все записи;
- посмотреть список доступных команд;
- выйти из приложения.

Выбор каждой команды будет осуществляться по нажатию соответствующей клавиши. Как вы, возможно, уже догадались, для реализации подобной логики нам потребуются операторы `switch` и главный цикл `while`. И так, обо всем по порядку.

Прежде чем реализовывать циклы, добавим переменную, в которой будет храниться выбранное действие пользователем (нажатая клавиша).

Удобнее всего использоваться тип `char`, в котором будет храниться один единственный символ. Добавляем строку:

```
char action = 'h';
```

Мы назвали переменную `action` (действие) и присвоили значение `'h'`. В будущем этот символ будет означать просмотр списка всех команд, это необходимо для того, чтобы при запуске программы первым делом отобразился список доступных команд.

Далее создадим наш главный цикл, в котором будет производиться ввод и выполнение команд.

```
while (action != 'q')
{
    // основной цикл
}
```

Изучив предыдущий материал, слушатель без труда определит, что подобный цикл `while` выполняется, пока в переменную `action` не записан символ `q`, другими словами, пока пользователь не нажмет клавишу `<q>` для выхода из приложения.

Теперь необходимо осуществить выполнение действий, соответствующих введенной команде. Для этого будем использовать оператор `switch`. Как вы помните, оператор `switch` включает один или несколько разделов переключения. Каждый раздел переключения содержит одну или несколько меток `case`, за которыми следует один или несколько операторов. Добавляем каркас нашего будущего ветвления. Пока обрабатываем нажатие только одной клавиши `<l>`. В примере ниже после нажатия на `<l>` ничего не произойдет, наполнение оператора будет добавлено позже.

```
switch (action)
{
    case 'l':
        // если action равен 'l' то выполнить описанные
        здесь операции
        break;
}
```

Раз мы добавили первую команду, – нажатие на клавишу `<l>`, давайте реализуем логику этого запроса. По нажатия на `<l>` мы будем вы-

водить для пользователя все записи. Для отображения записей выполним проход по списку, и вывод каждой записи в отдельную строку.

```
case '1':
    foreach (var note in notes)
    {
        Console.WriteLine((notes.IndexOf(note)+1) + ") " +note);
        // т.к. в списке нумерация начинается с 0, то прибавляем 1
    }
    break;
```

Вспомним материал из предыдущего раздела. **Foreach** – цикл, который проходит по всем записям в переменной **notes**, при этом в теле цикла обращение к текущей записи происходит через переменную **note**. Фактически на каждом шаге цикла переменная **note** смещается вправо, получает следующее значение, выводит его на экран и идет дальше (рис. 18).

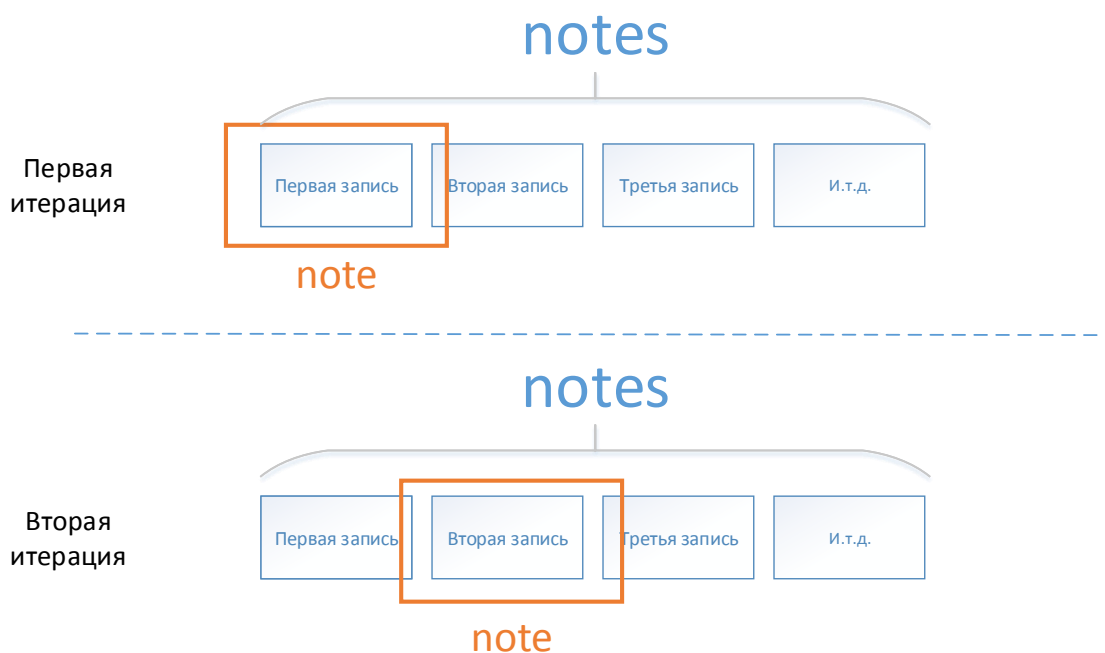


Рис. 18. Обход списка операцией `foreach`

Для вывода записи используется команда `Console.WriteLine`. Для получения номера записи используется команда `notes.IndexOf`. Фактически функция `IndexOf` объекта `notes` возвращает номер элемента списка, который был в нее передан. Стоит заметить, что в программе нумерация записей начинается с `0`, поэтому при выводе мы прибавляем к текущему номеру записи `1` с помощью вот такой конструкции

`IndexOf(note)+1`). Данная конструкция позволит выводить записи в формате: «1) Название элемент».

Далее определим операцию вывода доступных команд, мы договорим, что список доступных команд будет выводиться при старте программы и при нажатии пользователем клавиши **<h>**.

```
case 'h':
    Console.WriteLine("Доступные команды:");
    Console.WriteLine("a - добавить запись");
    Console.WriteLine("d - удалить запись с номером n");
    Console.WriteLine("l - список всех записей");
    Console.WriteLine("h - список доступных команд");
    Console.WriteLine("q - выйти из программы");
    break;
```

Реализуем следующую операцию – добавление новой записи. Пусть она выполняется при нажатии пользователем клавиши **<a>**.

```
case 'a':
    Console.Write("Введите сообщение: ");
    var newNote = Console.ReadLine(); // считываем сообщение
    notes.Add(newNote); // добавляем сообщение в конец списка
    break;
```

С помощью команды `Console.ReadLine` мы считываем сообщение, вводимое пользователем. А с командой `Add` мы уже знакомы, она добавила введённую пользователем строку в конец списка.

Теперь определим операцию удаления записи. Удаление будет происходить, если пользователь нажмет клавишу **<d>**.

```
case 'd':
    Console.Write("Введите номер записи для удаления: ");
    int n = Convert.ToInt32(Console.ReadLine())-1;
    if (n < notes.Count && n > -1)
    {
        notes.RemoveAt(n);
    } else
    {
        Console.WriteLine("Записи с указанным номером не существует");
    }
    break;
```

В данном блоке, пользователю предлагается ввести номер записи, которую он хочет удалить. Стоит обратить внимание на то, что команда `Console.ReadLine` возвращает в результате строку типа `string`, а нам нужен номер записи типа `int` (число), поэтому мы преобразуем строку типа `string` в номер типа `int` с помощью команды `Convert.ToInt32`.

Так же мы вычитаем 1 из введенного номера, поскольку нумерация записей в программе начинается с 0. Затем мы проверяем, что введенный номер принадлежит отрезку `[0; notes.Count]`, где `notes.Count` – команда для получения количества записей в списке. Если номер введен верно, то удаляем запись из списка с помощью команды `notes.RemoveAt`, иначе выводим сообщение об ошибке.

Если же ни одна из операций не была выполнена, значит пользователь ввел неизвестную команду. Чтобы оповестить его об этом используем секцию `default` оператора `switch`.

```
default:
    Console.WriteLine("Неизвестная команда");
    break;
```

На этом мы заканчиваем работу с оператором `switch`. Все возможные действия пользователя были обработаны.

Последнее, что остается сделать, это обеспечить ввод новой команды. Наверное, вы обратили внимание, что до сих пор мы не реализовали участок, который бы предлагал пользователю ввести команду. Добавим его сразу после блока `switch`. Обратите внимание, что блок ввода новой команды, тем не менее, должен находиться внутри цикла `while`, иначе программа завершит свою работу после выполнения первого действия.

```
Console.Write("Введите команду: ");
action = Console.ReadKey().KeyChar; // считываем новое действие
Console.WriteLine(); // для переноса вывода на новую строку
```

Команда `Console.ReadKey` в отличие от `Console.ReadLine` считывает нажатие одной клавиши, а метод `.KeyChar` позволяет получить символ нажатой клавиши, который мы и сравниваем в операторе `switch` описанном выше.

Используя псевдокод, структура приложения должна быть следующая:

```
функция Main
{
    Реализация хранилища

    Общий цикл While
    {
        Оператор switch
        {

        }

        Блок ввода новой команды
    }
}
```

2.3. Запуск программы

На этом реализация одной из самых простых версий справочника завершена. К сожалению, общий листинг программы приведен не будет. Результат выполнения программы изображен на рис. 19. При выполнении тестового задания студенты могут реализовать другие команды и использовать свои обозначения для каждой команды.

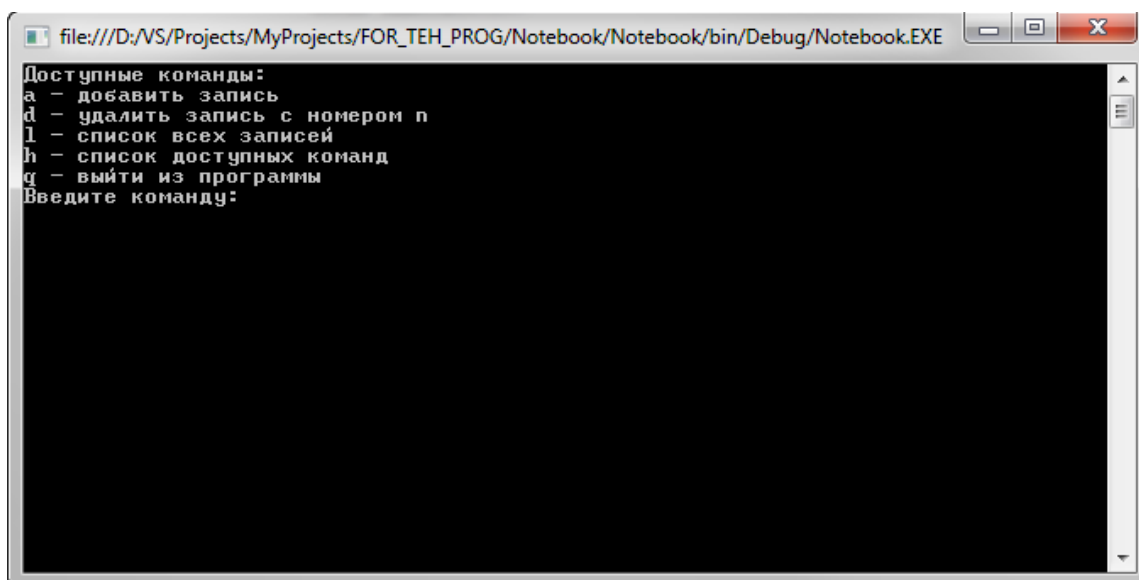


Рис. 19. Результат работы программы

2.4. Задание на лабораторную работу № 1

1. Реализовать консольное приложение в соответствии с вариантом.
2. Реализовать блок-схему основных элементов структурного программирования на основе примитивов, приведенных в методических указаниях.
3. Для ввода команд пользователем использовать цифры.
4. Добавить одну дополнительную команду, которая не была описана в методических указаниях.
5. Оформить отчет с описанием хода выполнения вашей лабораторной работы.

Номер варианта задания лабораторной работы № 1 определяется по последним двум цифрам номера зачетной книжки. Если образуемое ими число больше 12, то следует взять сумму этих цифр.

Например, если номер зачетной книжки Д-8В31/07, то номер варианта задания равен 7. Если номер зачетной книжки 3-8В31/23, то номер варианта задания равен 5 ($2+3=5$).

Варианты заданий

1. Телефонный справочник (в одной строке вводить: «ФИО, Телефон»).
2. Записная книжка (в одной строке вводить: «Запись, Дата»).
3. Список записей к врачу (в одной строке вводить: «Название врача, Дата»).
4. Список клиентов автомойки (в одной строке вводить: «Марка машины, Время»).
5. Список заказов в кафе (в одной строке вводить: «Номер стола, Блюдо»).
6. Путеводитель (в одной строке вводить: «Название места, Описание»).
7. Гостевая книга (в одной строке вводить: «Сообщение, Время»).
8. Почтовые данные (в одной строке вводить: «Адрес, Индекс»).
9. Список студентов (в одной строке вводить: «ФИО, Номер группы»).
10. Список рейсов (в одной строке вводить: «Название рейса, Дата прибытия»).
11. Список билетов на поезд (в одной строке вводить: «Номер вагона, Номер места»).
12. База автомобилей (в одной строке вводить: «Номер машины, Марка машины»).

Требования к оформлению отчета

1. Формат листа – А4 (210x297 мм). Поля: слева – 2,5 см; справа – 1,5 см; сверху, снизу – 2 см.
2. Основной текст документа должен быть напечатан шрифтом Times New Roman Cyr, размером 12 пт. Межстрочный интервал – 1,5 строки, абзацный отступ – 1,25 см. Не добавлять разрыв между абзацами одного стиля. Выравнивание текста – по ширине.
3. Формат заголовков.
Абзац: отступ – 1,25 см, выравнивание – по левому краю.
Шрифт:
 - заголовок первого уровня: размер – 12 пт, начертание – полужирный, регистр – все прописные;
 - заголовок второго и последующих уровней: размер – 12 пт, начертание – полужирный, регистр – как в предложении.
4. Нумерация страниц производится автоматически.
5. Если позволяет структура документа, автоматически оформить оглавление.

6. Титульный лист должен быть оформлен по стандарту СТО ТПУ 2.5.01–2006 (<http://portal.tpu.ru/departments/head/methodic/standart>).

7. Рекомендуемая структура отчета:

- титульный лист;
- оглавление (если требуется);
- введение;
- ход работы (описывается последовательность выполнения заданий, при необходимости приводятся участки кода и скриншоты).
- заключение.

Требования к защите

Лабораторная работа считается защищенной при наличии трех файлов:

1. Архив с исходным кодом.
2. Исполняемый файл.
3. Отчет по лабораторной работе.

3. ЛАБОРАТОРНАЯ РАБОТА № 2

ФУНКЦИОНАЛЬНЫЙ ПОДХОД

3.1. Ключевые особенности функционального подхода

Первыми формами модульности, появившимися в языках программирования, были процедуры и функции. Поскольку функции в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов программирования процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, – задачу повторного использования программного кода. Один раз написанную функцию можно многократно вызывать в программе с разными значениями параметров, передаваемых функции в момент вызова. Встроенные в язык функции позволяли существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из языка программирования.

С появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ОО-языков, к которым относится и язык С#, роль архитектурного модуля играет класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеет содержательную начинку, задавая некоторую абстракцию данных.

Процедуры и функции связываются теперь с классом, они обеспечивают требуемую функциональность класса и называются методами класса. Поскольку класс в объектно-ориентированном программировании рассматривается как некоторый тип данных, то главную роль в классе начинают играть его данные – поля класса, задающие свойства объектов класса. Методы класса «служат» данным, занимаясь их обработкой. Помните, в С# процедуры и функции существуют только как методы некоторого класса, они не существуют вне класса.

В данном контексте понятие класс распространяется и на все его частные случаи – структуры, интерфейсы, делегаты.

В языке С# нет специальных ключевых слов – `method`, `procedure`, `function`, но сами понятия, конечно же, присутствуют. Синтаксис объявления метода позволяет однозначно определить, чем является метод: процедурой или функцией.

3.2. Инициализация функции Main

Как вы могли заметить, в предыдущей лабораторной работе, весь код нашего приложения был написан в рамках одной единственной функции – `Main`. Функциональный подход предполагает разложение кода на ряд примитивов выполняющих определенное действие – функций. Таким образом, задачей третьей лабораторной работы будет вынесение простых пользовательских операций в функции.

Как вы считаете, какие основные плюсы функционального подхода? Во-первых, это примитивная декомпозиция кода на более мелкие куски, что позволяет упростить читабельность. Но, ключевое преимущество вынесения отдельных блоков в функции – это повторное использование кода. Так, используя структурный подход, нам бы пришлось каждый раз писать код добавляющий строку в массив или выполнять эту операция в цикле. Давайте исправим эту избыточность в третьей лабораторной работе.

Примечание. С остальными плюсами функционального подхода вы познакомитесь, прочитав лекционный материал курса.

Как уже было упомянуто ранее, тело всей программы во второй лабораторной работе находилось внутри одной функции `Main`. Вспомним, как выглядела наша функция, изучив код ниже.

```
static void Main(string[] args)
{
    Какой-то код
}
```

И так, приведенная выше строка инициализирует функцию `Main`. Как показано на рис. 20 **тело или сигнатура** функции состоит из следующих ключевых элементов: возвращаемый тип, имя функции, параметры или аргументы функции. Поле модификатор не относится непосредственно к функции, поэтому мы не будем его рассматривать, а лишь запомним, что в первых лабораторных работах, все функции будут иметь модификатор `static`.

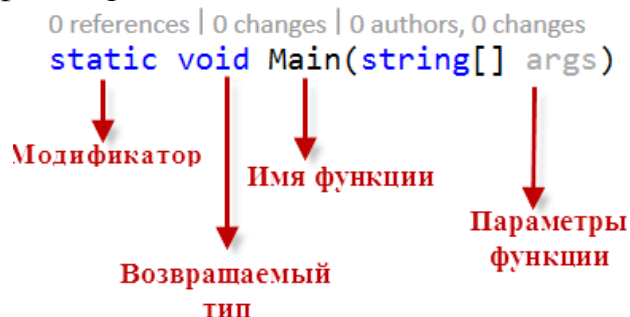


Рис. 20. Сигнатура (тело) функции

Важно!

Функция в языке C#, (как в прочем и любом другом языке программирования) отражает какой-либо процесс. Например, процесс сложения двух чисел можно назвать функцией **Add**, процесс вывода информации на экран **Print** и так далее. Стоит отметить, что неправильно будет называть функцию **Length** (Длина), так как слово **length** не отражает никакого процесса или действие. Верное название для подобной функции может быть, например, **GetLength** (Получить длину) или **SetLength** (Установить длину).

3.3. Возвращаемый тип

Каждая функция либо:

- возвращает что-то (**int Sum()**, **int GetLenght()**, **string GetName()**);
- ничего не возвращает (**void Main()**, **void Print()**, **void Draw()**, **void Format()**).

Каждая функция может либо возвращать внешнему миру (вызывающему коду) какие-то данные, либо просто выполнять какую-либо задачу. Предположим, у нас есть функция **Sum**, которая выполняет сложение двух чисел, очевидно, что такая функция должна вернуть число, равное этой сумме, тоже касается и функции **GetLength** (получить длину), которая возвращает число равное длине (например, длине диагонали вашего телевизора).

С другой стороны функция **Print()** просто выводит какие-то данные на консоль и ей не требуется ничего возвращать, функция **Drow()** может просто рисовать квадрат на холсте и ей тоже нечего вернуть внешнему миру. Наверно многим уже стало понятно, что при работе с функциями, которая ничего не возвращает, используется ключевое слово **void** (пустота), а для функций с возвращаемым значением указывается необходимый тип данных (**int**, **string** и др.).

3.4. Имя функции

Мы не будем подробно останавливаться на данной части сигнатуры функции – очевидно, что имя функции используется для дальнейшего обращения к созданной функции по коду проекта. Стоит отметить, что все функции должны начинаться с большой буквы (пример: **Main**, **Sum**, **Print**, **Drow**). Если имя функции состоит из двух слов, используется стиль написания **CamelCase**, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы.

Примечание. Стилль получил название CamelCase, поскольку заглавные буквы внутри слова напоминают горбы верблюда (например: DrawImage, PrintName).

3.5. Параметры функции

Если вы уже читали лекционный материал курса, то можете помнить, что список формальных аргументов метода может быть пустым. Хотя в случае с функцией `Main` это не так. Пустой список параметров или аргументов – довольно типичная ситуация. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Рассмотрим синтаксис объявления одного формального аргумента:

`[ref|out|params]` **тип_аргумента** **имя_аргумента**

Обязательным является указание типа и имени аргумента. Обратите внимание, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, классом, структурой, интерфейсом, перечислением, функциональным типом.

Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему **произвольное число фактических аргументов**. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово `params`. Оно может появляться в объявлении лишь для последнего аргумента списка, объявляемого как массив произвольного типа. При вызове метода этому формальному аргументу соответствует произвольное число фактических аргументов.

Содержательно, все аргументы метода разделяются на три группы: **входные, выходные и обновляемые**. Аргументы первой группы передают информацию методу, их значения в теле метода только читаются. Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода. Аргументы третьей группы выполняют обе функции. Их значения используются в ходе вычислений и обновляются в результате работы метода. Выходные аргументы всегда должны сопровождаться ключевым словом `out`, обновляемые – `ref`. Что же касается входных аргументов, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром `ref`, о чем мы будем говорить позже. Обратите внимание, если аргумент объявлен как выходной с ключевым словом `out`, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

3.6. Тело функции

Синтаксически тело метода является блоком, представляющим последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор, возвращающий значение функции в форме `return <выражение>`.

Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.

3.7. Структуры

В данной лабораторной работе, помимо знакомства с функциями, студентам предлагается реализовать хранилище данных в виде массива структур. Ниже дан краткий обзор понятия структуры в языке C#.

В .NET. Типы структур хорошо подходят для моделирования математических, геометрических и других «атомарных» сущностей в приложении (в нашем случае структура будет моделировать объект – Запись). Структура – это определяемый пользователем тип; структуры представляют собой типы, которые могут содержать любое количество полей данных и членов, оперирующих над этими полями.

Примечание. Если вы имеете опыт объектно-ориентированного программирования, можете считать структуры «облегченными классами», т.к. они предоставляют способ определения типа, поддерживающего инкапсуляцию, но не могут применяться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных через наследование, необходимо использовать классы.

На первый взгляд процесс определения и использования структур выглядит очень просто, но, как известно, сложности обычно скрываются в деталях. В C# структуры определяются с применением ключевого слова **struct**. Определим новую структуру под названием **Note**, содержащую две переменных-члена типа **string** и **DateTime** и набор методов для взаимодействия с ними.

```

class Program
{
    struct Note
    {
        public string text;
        public DateTime date;

        public Note(string note)
        {
            text = note;
            date = DateTime.Now;
        }
    }
}

```

До этого, как вы помните, наше приложение позволяло хранить запись в виде строки символов. Предположим, что мы хотим иметь возможность хранить помимо самой записи время ее добавления. Для этого мы могли бы создать дополнительные переменные-списки, однако, это значительно усложнит нашу программу, поскольку мы будем вынуждены контролировать, чтобы номер записи в списке соответствовал номеру даты в списке и т.д. Но с помощью структуры мы можем в одном списке, в каждой записи хранить несколько значений. Теперь каждый элемент нашего списка будет хранить структуру **Note**, которая в свою очередь хранит информацию о дате и само сообщение. Надеюсь, вы не запутались во вложенности.

Примечание. Код структуры должен располагаться до функции Main.

3.7.1. Конструктор структуры

Для того чтобы каждый раз во время добавления не задавать вручную дату, создадим конструктор, который будет автоматически записывать дату в поле **date**. Заметьте, что поля и конструктор объявлены с модификатором **public**, для того чтобы к ним можно было обратиться извне.

Конструктор структуры – это специальная функция, которая не имеет типа возвращаемого значения и не помечается ключевым словом **void**, фактически эта функция вызывается единожды, при создании структуры и служит некой отправной точкой. В рамках конструктора обычно производят различные настройки, в нашем случае, мы выполняем инициализацию двух переменных.

```

    public Note(string note)
    {
        text = note;
        date = DateTime.Now;
    }

```

3.8. Модернизация приложения «Записная книжка»

После небольшого теоретического обзора модернизируем нашу лабораторную работу. Во-первых, необходимо отредактировать объявление списка записей.

```
List<Note> notes = new List<Note>();
```

Обращаю ваше внимание, что теперь список содержит объекты **Note**. В связи с этим изменим и код добавления записей.

```
notes.Add(new Note("Первая запись"));
notes.Add(new Note("Вторая запись"));
```

На данный момент наше приложение не использует возможности функций, не считая функцию **Main**. Поскольку функции могут использоваться не только для сокращения повторяющихся участков кода, но и для повышения читаемости кода, мы вынесем наши операции в блоке **switch** в отдельные функции.

Как мы уже говорили, в **C#** наименования функций принято оформлять в **Pascal case** стиле, когда каждое слово начинается с заглавной буквы, например, «**GetNewMessage**».

Вынесем в отдельную функцию код, отвечающий за вывод списка записей.

```
static void NotesList(List<Note> notes)
{
    foreach (var note in notes)
    {
        Console.WriteLine((notes.IndexOf(note) + 1) + ") " + note.text + " (" +
            note.date.ToString("dd MMM HH:mm:ss") + ")"); // т.к. в списке нумерация начина-
            ется с 0, то прибавляем 1
    }
}
```

Заметьте, что при выводе даты необходимо задать формат. В нашем случае это день, месяц, часы, минуты и секунды.

В качестве аргумента мы передаем наш список записей. Стоит заметить, что хоть аргумент и имеет имя **notes**, он вовсе не привязан к переменной **notes** в главной функции **Main**. Другими словами, мы можем заменить **notes** на любое другой доступное имя переменной, правда, тогда придется отредактировать тело функции.

Заметьте, что для вывода индекса записи использована конструкция «**notes.IndexOf(note) + 1**». Студентам можно создать дополнительное поле в структуре с номером записи.

Вынесем функционал нашей программы в отдельные функции. А именно:

Функция отображения справки:

```
static void ShowHelp()
{
    Console.WriteLine("Доступные команды:");
    Console.WriteLine("a - добавить запись");
    Console.WriteLine("d - удалить запись с номером n");
    Console.WriteLine("h - список доступных команд");
    Console.WriteLine("l - список всех записей");
}
```

Функция добавления записи:

```
static void AddNote(List<Note> notes)
{
    Console.Write("Введите сообщение: ");
    var newNote = Console.ReadLine(); // считываем сообщение
    notes.Add(new Note(newNote)); // добавляем сообщение в конец списка
}
```

Функция удаления записи:

```
static void DeleteNote(List<Note> notes)
{
    Console.Write("Введите номер записи для удаления: ");
    int n = Convert.ToInt32(Console.ReadLine()) - 1;
    // проверим существует ли введенный номер записи
    if (n < notes.Count && n > -1)
    {
        notes.RemoveAt(n);
    }
    else
    {
        Console.WriteLine("Записи с указанным номером не существует");
    }
}
```

Самостоятельно изучите структуру функций, обращая внимание на синтаксис обращения к полям структуры `Note`.

Финальной частью является изменение блока `switch`. Теперь в каждом узле будут вызываться соответствующие функции. Финальный вариант блока `switch` приведен ниже.

```

switch (action)
{
    // вывести все записи
    case 'l':
        NotesList(notes);
        break;

    case 'h':
        ShowHelp();
        break;

    // добавить запись
    case 'a':
        AddNote(notes);
        break;

    // удалить запись
    case 'd':
        DeleteNote(notes);
        break;

    default:
        Console.WriteLine("Неизвестная команда");
        break;
}

```

Как мы видим, блок `switch` стал более лаконичным. Читаемость кода значительно возрастает и ко всему прочему, созданные выше функции, можно будет использовать в дальнейшем, что значительно сократит дублирование кода.

На этом переход на функциональный или процедурный подход завершен. Код программы состоит из набора функций и операторов вызова данных функций.

Лабораторная работа № 2 завершена.

3.9. Задание на лабораторную работу № 2

1. Модернизировать консольное приложение, созданное в лабораторной работе № 1, путем выноса основных операций в функции.
2. Функции должны иметь осмысленные названия, т.е. отображающие суть функций.
3. Создать структуру, содержащую четыре поля.
4. Заменить изначальный список на список структур.
5. Оформить отчет с описанием хода выполнения вашей лабораторной работы.

Требования к оформлению отчета

1. Формат листа – А4 (210х297 мм). Поля: слева – 2,5 см; справа – 1,5 см; сверху, снизу – 2 см.
2. Основной текст документа должен быть напечатан шрифтом Times New Roman Cyr, размером 12 пт. Межстрочный интервал – 1,5 строки, абзацный отступ – 1,25 см. Не добавлять разрыв между абзацами одного стиля. Выравнивание текста – по ширине.
3. Формат заголовков.
Абзац: отступ – 1,25 см, выравнивание – по левому краю.
Шрифт:
 - заголовок первого уровня: размер – 12 пт, начертание – полужирный, регистр – все прописные;
 - заголовок второго и последующих уровней: размер – 12 пт, начертание – полужирный, регистр – как в предложении.
4. Нумерация страниц производится автоматически.
5. Если позволяет структура документа, автоматически оформить оглавление.
6. Титульный лист должен быть оформлен по стандарту СТО ТПУ 2.5.01–2006 (<http://portal.tpu.ru/departments/head/methodic/standart>).
7. Рекомендуемая структура отчета:
 - титульный лист;
 - оглавление (если требуется);
 - введение;
 - ход работы (описывается последовательность выполнения заданий, при необходимости приводятся участки кода и скриншоты);
 - заключение.

Требования к защите

Лабораторная работа считается защищенной при наличии трех файлов:

1. Архив с исходным кодом.
2. Исполняемый файл.
3. Отчет по лабораторной работе.

4. ЛАБОРАТОРНАЯ РАБОТА № 3.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

На сегодняшний день писать что-либо об объектно-ориентированном программировании, по мнению автора, является плохим тоном. Ни для кого не секрет, что существует множество книг, статей и «шаг за шагом» инструкций, которые дают детальное описание всех ключевых особенностей ООП. Несмотря на обилие информации по данной тематике у слушателей курса зачастую возникает множество вопросов касающихся, как проектирование самих пользовательских типов – классов, так и базовых принципах, играющих важнейшую роль в построении объектно-ориентированных программных компонент.

После описания основ определения классов и размещения объектов в остальной части методического указания будут рассмотрены темы инкапсуляции, наследования и полиморфизма. По ходу работы с данным методическим указанием студенты узнают, как определяются свойства класса, смысл ключевых слов `private`, `public` и `protected`, познакомятся с понятием полиморфных интерфейсов другой полезной для выполнения лабораторных работ информацией.

4.1. Ключевые особенности ООП

В платформе .NET, наиболее фундаментальной программной конструкцией является тип класса. Формально класс – это определяемый пользователем тип, который состоит из полей данных (часто именуемых переменными-членами) и членов, оперирующих этими данными (конструкторов, свойств, методов, событий и т.п.).

Все вместе поля данных класса представляют «состояние» экземпляра класса (иначе называемого объектом). Еще никто не запутался? Давайте сразу разберем небольшой пример. Предположим, в процессе реализации программы разработчик столкнулся с задачей, которая требует манипуляции с неким объектом – автомобилем. Очевидно, что стандартные библиотеки .NET не имеют типа данных «Автомобиль» (а жаль). Попытка написать следующий код приведет к ошибке на этапе компиляции:

```
Car myCar;  
int newInt;
```

Проблема в первой строке, компилятору ничего не известно о Car, а вот строка `int newInt` не вызывает ни каких вопросов ни у компилятора, ни у читателей (надеюсь). Таким образом, для решения поставленной задачи класс Car сначала надо реализовать – строго говоря, добавить в систему новый тип данных. Хотелось бы заметить, что **класс и экземпляр класса (он же объект) это разные понятия**, которые не следует путать. Проводя аналогии, можно назвать класс неким шаблоном, заготовкой, которая будет использована для создания реально хранящихся в памяти экземпляров или объектов класса со своим уникальным состоянием. На примере автомобиля уникальное состояние может характеризоваться VIN номером, текущей скоростью или фамилией владельца. Все ключевые особенности ООП будут еще раз продублированы ниже, по ходу изложения материала и мы еще не раз к ним вернемся.

Подводя промежуточный итог можно сказать, что мощь объектно-ориентированных языков, подобных C#, состоит в их способности группировать данные и связанную с ними функциональность в определении класса, что позволяет моделировать программное обеспечение на основе сущностей реального мира (автомобилей, людей, животных и пр.).

4.2. Первый класс

Попробуем реализовать первый класс и разобрать его ключевые компоненты. Для начала создадим простое консольное приложение, для тех, кто забыл, как это сделать, пример представлен на рис. 21.

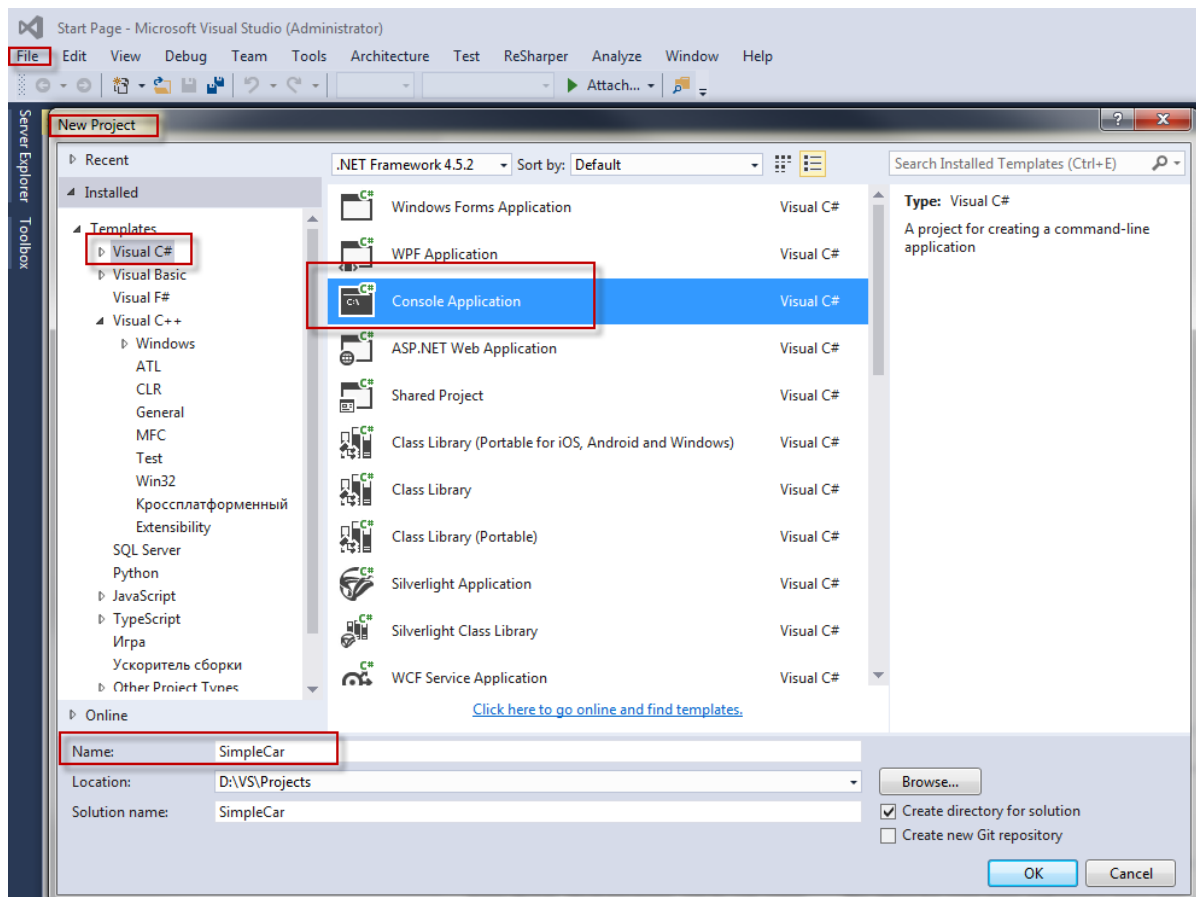


Рис. 21. Создание консольного приложения

После того, как проект был успешно создан, добавим в него новый класс `Car`. Для того, чтобы это сделать достаточно написать следующий очень простой код.

```
class Car
{
}
```

Обратите внимание, что объявление класса `Car` должно быть за пределами класса `Program`. На самом деле реализацию новых классов лучше выполнять в отдельных файлах, это делает структуру программы более читабельной и прозрачной. Для того, чтобы вынести наш класс `Car` в отдельный файл – создадим новый файл, используя меню *Project -> Add Class...* (рис. 22).

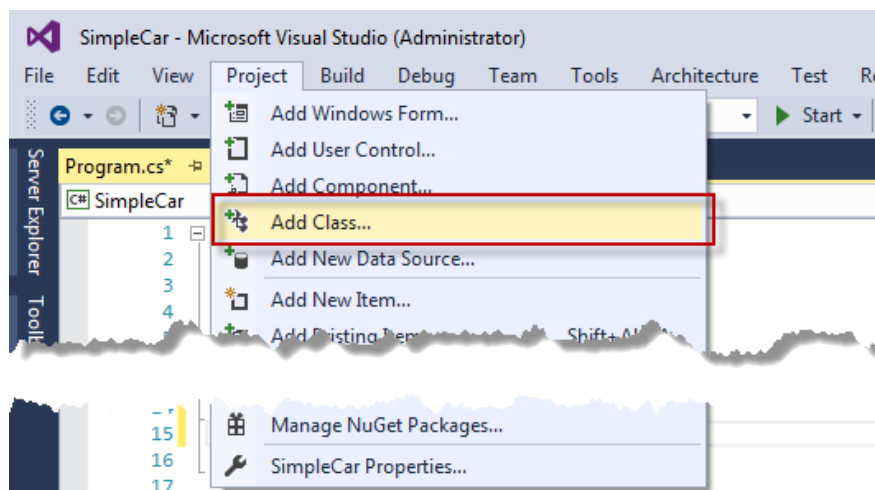


Рис. 22. Создание нового файла для хранения класса

Помощник Visual Studio предложит выбрать нужный тип файла (выбираем Class) и ввести название. Название файла должно (рекомендуется) быть идентично названия класса, в нашем случае Car (рис. 23).

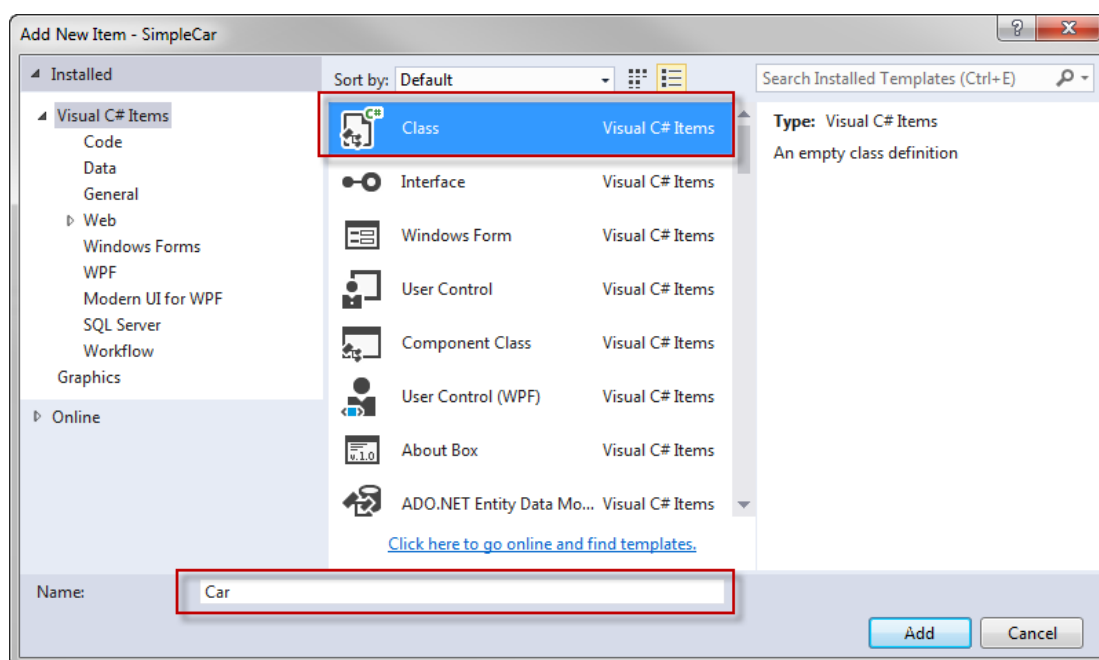


Рис. 23. Новый файл Car.cs

В обозревателе решения (Solution Explorer) будет отображен новый файл Car.cs, в котором уже создан новый класс Car (Visual Studio выполнила объявление элементарного класса за нас, спасибо ей) (рис. 24).

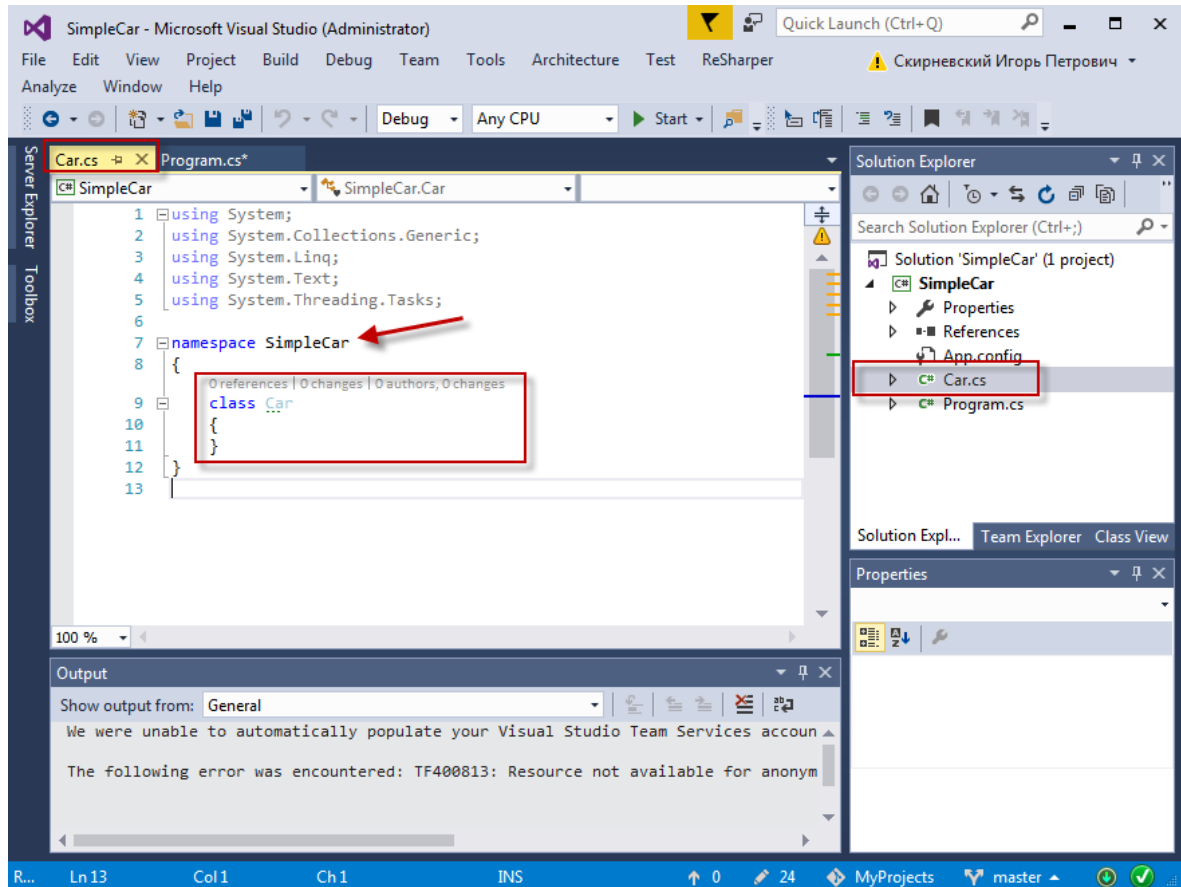


Рис. 24. Дерево проекта с добавленным файлом Car.cs

Стрелкой обозначено пространство имен **SimpleCar**, это говорит о том, что класс **Car**, хоть и вынесен в отдельный файл, находится в том же пространстве имен, что и функция **Main** из класса **Program** (файл **Program.cs**).

После определения типа класса нужно определить набор переменных-членов, которые будут использоваться для представления его состояния. Например, вы можете решить, что объекты **Car** (автомобили) должны иметь поле данных типа **int**, представляющее текущую скорость, и поле данных типа **string** для представления VIN номера автомобиля.

С учетом этих начальных проектных положений класс **Car** будет выглядеть следующим образом:

```
class Car
{
    // 'Состояние объекта Car.
    public string vin;
    public int currSpeed; }
```

Обратите внимание, что переменные-члены объявлены с использованием модификатора доступа **public**. Открытые (**public**) члены класса доступны непосредственно, как только создается объект данного типа. Термин «объект» служит для представления экземпляра данного типа класса, созданного с помощью ключевого слова **new**.

Информация: Поля данных класса редко (если вообще когда-нибудь) должны определяться с модификатором **public**. Чтобы обеспечить целостность данных состояния, намного лучше объявлять данные закрытыми (**private**) или, возможно, защищенными (**protected**) и разрешать контролируемый доступ к данным через свойства (как будет показано далее). Однако, чтобы сделать первый пример насколько возможно простым, мы оставляем поля данных открытыми.

После определения набора переменных-членов, представляющих состояние класса, следующим шагом в проектировании будет создание членов, которые моделируют его поведение. Для этого примера класс **Car** определяет один метод по имени **SpeedUp ()** и еще один – по имени **PrintState ()**.

Внимательный читатель мог заметить различие между переменными-членами и методами класса. По сути, переменная-член это обычная переменная, с которой мы встречались в предыдущих лабораторных работах, которая «находится внутри класса» и отражает его состояние. Если проводить аналогии, то переменную-член можно сравнить с параметрами объекта (массой, давлением, размером и пр.). Важно также обратить внимание на то, что переменные-члены принято называть именами существительными (скорость, номер, имя).

Методы в свою очередь – это функции находящиеся внутри класса, обычно они выражены глаголами: Отобразить скорость, получить имя, увеличить давление (**GetSpeed**, **SetName** и т.д.).

Наш класс уже содержит несколько переменных-членов, давайте добавим в него ряд методов. Модифицируйте код класса следующим образом:

```
class Car
{
    // 'Состояние объекта Car.
    public string petName;
    public int currSpeed;

    // Функциональность Car.
    public void PrintState()
    {
        Console.WriteLine("{0} is going {1} MPH.", vin, currSpeed);
    }
    public void SpeedUp(int delta)
```

```

    {
        currSpeed += delta;
    }
}

```

Метод `PrintState ()` – это в какой-то мере диагностическая функция, которая просто выводит текущее состояние объекта `Car` в окно командной строки. Метод `SpeedUp ()` повышает скорость `Car`, увеличивая ее на величину, переданную во входящем параметре типа `int`.

После того, как наш класс объявлен, в него добавлены методы и переменные-члены, самое время создать экземпляр данного класса и посмотреть, на что он способен. Модифицируем метод `Main` в классе `Program`, добавив новый объект (экземпляр) класса `Car`.

```

// Разместить в памяти и сконфигурировать объект Car.
Car myCar = new Car();
myCar.vin = "10BDCAR";
myCar.currSpeed = 10;
// Увеличить скорость автомобиля в несколько раз и вывести новое состоя-
ние,
for (int i = 0; i <= 10; i++)
{
    myCar.SpeedUp(5);
    myCar.PrintState();
}
Console.ReadLine();

```

Обратите внимание на строку `Car myCar = new Car();`. Давайте разберем ее более детально. Первая часть (левая часть) содержит имя нашего нового типа `Car` и название нашей переменной `myCar` этого типа. После оператора присваивания под новую переменную выделяется память через оператор `new` и вызывается конструктор класса `Car()`. О конструкторах мы поговорим позже. Именно в этой строке происходит создание экземпляра класса, реального объекта, который будет храниться в памяти и которым можно манипулировать (менять его состояние).

Мы не будем подробно рассматривать результаты работы приведенного выше кода, вы сможете проделать это самостоятельно. Проведите несколько экспериментов с заданием `vin`-а или скорости, а также вызовом методов класса.

4.3. Ключевое слово `new`

Как было показано в предыдущем примере, объекты должны быть размещены в памяти с использованием ключевого слова `new`. Если ключевое слово `new` не указать и попытаться воспользоваться переменной

класса в следующем операторе кода, будет получена ошибка компиляции. Например, следующий метод `Main ()` компилироваться не будет:

```
static void Main(string[] args)
{
    Console.WriteLine("***** FunwithClassTypes *****\n");
    // Ошибка! Забыли использовать new для создания объекта!
    Car myCar;
    myCar.petName = "Fred";
}
```

Все дело в том, что объект `myCar` еще не размещен в памяти. Да, фактически система уже знает о том, что имя `myCar` зарезервировано под будущий экземпляр класса, т.е. `myCar` определен, но еще не создан (что-то вроде «Привет, я `myCar`, но меня еще не создали»). Чтобы корректно создать объект с применением ключевого слова `new`, можно определить и разместить в памяти объект `Car` в одной строке кода. Прodelайте это самостоятельно.

В качестве альтернативы, определение и размещение в памяти экземпляра класса может осуществляться в разных строках кода:

```
static void Main(string[] args)
{
    Console.WriteLine("***** FunwithClassTypes *****\n");
    // Ошибка! Забыли использовать new для создания объекта!
    Car myCar;
    myCar = new Car();
    myCar.petName = "Fred";
}
```

Здесь первый оператор кода просто объявляет ссылку на еще не созданный объект типа `Car`. Ссылка будет указывать на действительный объект в памяти только после явного присваивания.

В любом случае, к этому моменту мы получили простейший тип класса, который определяет несколько элементов данных и некоторые базовые операции. Теперь, чтобы расширить функциональность текущего класса `Car`, необходимо разобраться с ролью **конструкторов**.

4.4. Понятие конструкторов

Мы не будем детально останавливаться на всех особенностях конструкторов, которые существуют в .NET, многие важные темы, такие как главный конструктор, цепочка конструкторов, статический конструктор и др. останутся за пределами данного методического пособия и останутся для самостоятельного изучения слушателями курса.

Учитывая, что объект имеет состояние (представленное значениями его переменных-членов), обычно желательно присвоить осмыслен-

ные значения полям объекта перед тем, как работать с ним. В настоящий момент тип `Car` требует присваивания значений полям `vin` и `currentSpeed`. Для текущего примера это не слишком проблематично, поскольку открытых элементов данных всего два. Однако нередко классы состоят из нескольких десятков полей. Ясно, что было бы нежелательно писать 20 операторов инициализации для всех 20 элементов данных такого класса.

К счастью, в C# поддерживается механизм конструкторов, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор – это специальный метод класса, который вызывается неявно при создании объекта с использованием ключевого слова `new`. Однако в отличие от «нормального» метода, конструктор **никогда не имеет возвращаемого значения (даже `void`) и всегда именуется идентично имени класса, который он конструирует**. Еще раз перечитайте последнее предложение, это очень важная информация, которую следует запомнить.

4.4.1. Стандартный конструктор

Каждый класс C# снабжается стандартным конструктором, который при необходимости может быть переопределен. По определению стандартный конструктор никогда не принимает аргументов. После размещения нового объекта в памяти стандартный конструктор гарантирует установку всех полей данных в соответствующие стандартные значения (стандартные значения для типов данных C# по большому счету являются очевидными, нули для целых чисел, пустые строки для строк и так далее).

Если вы не удовлетворены такими стандартными присваиваниями, можете переопределить стандартный конструктор в соответствии со своими нуждами. В целях иллюстрации модифицируем класс C#, как показано ниже:

```
class Car
{
    // 'Состояние объекта Car.
    public string petName;
    public int currSpeed;

    // Специальный стандартный конструктор
    public Car()
    {
        vin = "0000CAR";
        currSpeed = 10;
    }
    ...
}
```

В данном случае мы заставляем объекты `Car` начинать свою жизнь под именем `Chuck` и скоростью 10 миль в час. При этом создавать объекты `Car` со стандартными значениями можно следующим образом:

```
Car chuck = new Car();
```

4.4.2. Определение специальных конструкторов

Обычно, помимо стандартного конструктора в классах определяются дополнительные конструкторы. При этом пользователь объекта обеспечивается простым и согласованным способом инициализации состояния объекта непосредственно в момент его создания. Следующее изменение класса `Car`, который теперь поддерживает целых три конструктора:

```
class Car
{
    // 'Состояние объекта Car.
    public string petName;
    public int currSpeed;

    // Специальный стандартный конструктор
    public Car()
    {
        petName = "Chuck";
        currSpeed = 10;
    }
    // Здесь currSpeed получает стандартное значение для типа int (0) .
    public Car(string pn)
    {
        petName = pn;
    }
    // Позволяет вызывающему коду установить полное состояние Car.
    public Car(string pn, int cs)
    {
        petName = pn;
        currSpeed = cs;
    }
    ...
}
```

Имейте в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) количеством и типом аргументов. При выполнении предыдущих лабораторных работ слушатели уже знакомились с понятиями перегруженных функций, и знают, что определение методов с одним и тем же именем, но разным количеством и типами аргументов, называется перегрузкой. Таким образом, класс `Car` имеет перегруженный конструктор, чтобы предоставить несколько способов создания объекта во время объявления. В любом случае, теперь можно со-

здавать объекты `Car`, используя любой из его открытых конструкторов. Попробуйте провести эксперименты с созданием экземпляра класса `Car` через разные конструкторы.

Как показано на рис. 25, среда Visual Studio сама подскажет вам, какие варианты конструкторов вы можете вызвать, и какие входные аргументы будет необходимо в них передать.

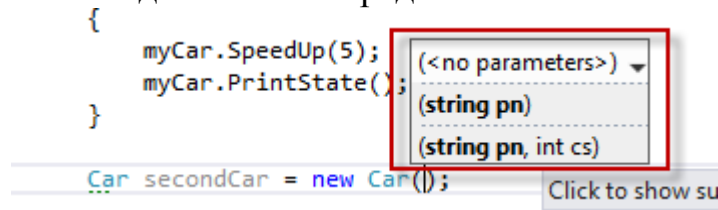


Рис. 25. Подсказка при использовании перегруженных конструкторов

Множество важных тем, таких как цепочка конструкторов, ключевое слово `this`, статический конструктор и др. останутся за пределами данных методических указаний, и читателям рекомендуется изучить данные темы самостоятельно. Также весьма важной темой, которая остается для самостоятельного изучения является создание статического класса, для которого невозможно создание экземпляров.

4.5. Основные принципы объектно-ориентированного программирования

Все объектно-ориентированные языки (C#, Java, C++, Smalltalk, Visual Basic и т.п.) должны отвечать трем основным принципам объектно-ориентированного программирования (ООП), которые перечислены ниже.

1. Инкапсуляция. Как данный язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
2. Наследование. Как данный язык стимулирует многократное использование кода?
3. Полиморфизм. Как данный язык позволяет трактовать связанные объекты сходным образом?

Прежде, чем погрузиться в синтаксические детали реализации указанных принципов, важно понять базовую роль каждого из них. При выполнении любой задачи с использованием объектно-ориентированного языка разработчику постоянно приходится к представленным выше принципам, чтобы значительно упростить (хотя в некоторых случаях и усложнить) и обезопасить свой код.

4.5.1. Роль инкапсуляции

Первый основной принцип ООП называется инкапсуляцией. Этот принцип касается способности языка скрывать излишние детали реализации от пользователя объекта. Например, предположим, что используется класс по имени `DatabaseReader`, который имеет два главных метода: `Open ()` и `Close ()`.

```
// Этот класс инкапсулирует детали открытия и закрытия базы данных.  
DatabaseReader dbReader = new DatabaseReader();  
dbReader.Open(@"C:\AutoLot.mdf");  
  
// Сделать что-то с файлом данных и закрыть файл.  
dbReader.Close();
```

Фиктивный класс `DatabaseReader` инкапсулирует внутренние детали нахождения, загрузки, манипуляций и закрытия файла данных. Программистам нравится инкапсуляция, поскольку этот принцип ООП упрощает кодирование. Нет необходимости беспокоиться о многочисленных строках кода, которые работают «за кулисами», чтобы реализовать функционирование класса `DatabaseReader`. Все, что потребуется – это создать экземпляр и отправить ему соответствующие сообщения (например, «открыть файл по имени `AutoLot.mdf`, расположенный на диске `C:`»). Таким образом, инкапсуляция – это сокрытие деталей реализации класса от внешнего мира и предоставление удобного интерфейса взаимодействия.

Давайте рассмотрим пример. Человек планирует выпить чашечку горячего кофе с утра у себя дома, он подносит стакан и нажимает на кнопку «капучино» – и это все, больше ни каких операций. Проведем аналогии, кофе машина в данном случае инкапсулирует, т.е. скрывает от пользователя множество операций, таких как помол кофе, набор кипятка, варка, налив кофе в стакан и пр. С другой стороны, кофе машина предоставляет удобный интерфейс в виде кнопки «капучино», который за кулисами (внутри класса) запускает нужную последовательность шагов. Очевидно, если бы для пользователя была предоставлена полная свобода, то он бы мог нарушить что-то в цикле варки и нанести вред как аппарату, так и себе.

В приведенном выше примере класс `DatabaseReader` тоже скрывает весь алгоритм подключения (а он может быть весьма нетривиальным) от пользователя и предоставляет только одну кнопку «`Open`». Надеюсь данный пример поможет вам понять первый принцип ООП – инкапсуляцию данных.

Забегаая вперед, при реализации класса избегайте открытый переменных-членов, данные поля данных всегда должны быть закрыты (**private**, **protected**) и доступ к ним должен быть предоставлен через методы или свойства. Для доказательства логичности данного утверждения может привести небольшой пример: возраст человека является переменной-членом класса Человек и доступ к данной информации осуществляется через метод «Спросить возраст», который в зависимости от контекста может сообщить истинный возраст, а может вернуть ошибку «отказано в доступе». Думаю, на земле найдется не так много людей, которые выставляют на показ данные своих личных счетов, паролей, личную информацию о себе и пр. Для получения информации подобного рода требуется обратиться к методам «Спросить...». Все это еще раз подчеркивает связь ООП с реальным миром.

Соккрытие или открытие полей класса происходит за счет модификаторов доступа, с которыми мы уже сталкивались. В табл. 2 приведено описание всех существующих в языке C# модификаторов доступа.

Таблица 2

Модификаторы доступа C#

Модификатор доступа	К чему может быть применен	Назначение
public	Типы или члены типов	Открытые (public) элементы не имеют ограничений доступа. Открытый член может быть доступен как из объекта, так и из любого производного класса. Открытый тип может быть доступен из других внешних сборок
private	Члены типов или вложенные типы	Закрытые (private) элементы могут быть доступны только в классе (или структуре), в котором они определены
protected	Члены типов или вложенные типы	Защищенные (protected) элементы могут использоваться классом, который определил их, и любым дочерним классом. Однако защищенные элементы не доступны внешнему миру через операцию точки (.)
internal	Типы или члены типов	Внутренние (internal) элементы доступны только в пределах текущей сборки. Таким образом, если в библиотеке классов .NET определен набор внутренних типов, то другие сборки не смогут ими пользоваться
protected internal	Члены типов или вложенные типы	Когда ключевые слова protected и internal комбинируются в объявлении элемента, такой элемент доступен внутри определяющей его сборки, определяющего класса и всех его наследников

4.5.2. Роль наследования

Следующий принцип ООП – наследование – касается способности языка позволять строить новые определения классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или родительского) класса, наследуя его основную функциональность в производном подклассе (также именуемом дочерним классом). На рис. 26 показан простой пример.

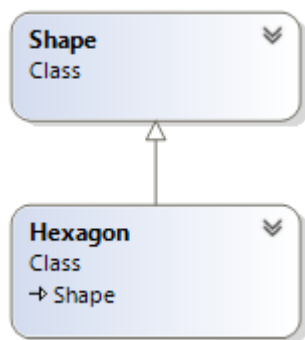


Рис. 26. Диаграмма классов, отражающая наследование

Прочесть диаграмму на рисунке 6 можно так: «шестиугольник (Hexagon) является фигурой (Shape)». При наличии классов, связанных этой формой наследования, между типами устанавливается отношение «является» (is-a). Такое отношение и называют наследованием. Здесь можно предположить, что **Shape** определяет некоторое количество членов, общих для всех наследников (скажем, значение для представления цвета фигуры и другие значения, задающие высоту и ширину). Учитывая, что класс **Hexagon** расширяет **Shape**, он наследует основную функциональность, определенную классом **Shape**, а также определяет дополнительные собственные детали, касающиеся шести угольников (какими бы они ни были).

Фактически, если у класса **Shape** есть метод **GetColor()**, то данный метод будет и у его наследника (передается ему по наследству), но только при условии, что модификатор доступа у метода в базовом классе будет **public** или **protected**. Все методы и переменные-члены помеченные, как **private** являются сокрыты даже от наследников.

Вначале данного методического пособия мы рассматривали пример с автомобилем, в нашем случае наследование может быть применено в случаи создание некоего обобщения над машиной – транспортного средства, родителя всего, что может ехать. Реализуем класс **Vehicle** (транспортное средство), код представлен ниже.

```

class Vehicle
{
    public Vehicle()
    {
        vin = "000CAR";
        currSpeed = 10;
    }
    public Vehicle(string pn)
    {
        vin = pn;
    }
    // Позволяет вызывающему коду установить полное состояние Car.
    public Vehicle(string pn, int cs)
    {
        vin = pn;
        currSpeed = cs;
    }

    protected string vin;
    protected int currSpeed;

    public void PrintState()
    {
        Console.WriteLine("{0} is going {1} MPH.", vin, currSpeed);
    }
    public void SpeedUp(int delta)
    {
        currSpeed += delta;
    }
}

```

Обратите внимание, что реализация класса **Vehicle** очень похожа на прошлую реализацию класса **Car**, теперь создадим класс и сделаем его наследником класса **Vehicle**.

```

class Car : Vehicle
{
    public Car()
    {
    }
    public Car(string pn, int cs = 5)
        :base(pn, cs)
    {
    }
}

```

Очевидно, что класс **Car** выглядит значительно более лаконичным. Он обладает всеми свойствами класса **Vehicle** и может быть расширен своими собственными. Что особенного есть у машины, чего может не быть у всех остальных типов транспортных средств, например, мотоцикла или велосипеда. Пусть это будет состояние подушек безопасности: работает или нет: `bool isAirbagOk`;


```

class Car : Vehicle
{
    public Car()
    {
    }
    public Car(string pn, int cs = 5)
        :base(pn, cs)
    {
    }
    public bool isAirbagOk = true;
}

```

Обратите внимание, что модификатор доступа для `isAirbagOk` установлен `public` – это не верно, в данном случае это нужно только для упрощения кода. Создадим экземпляр класса `Car` в функции `main` (рис. 27).

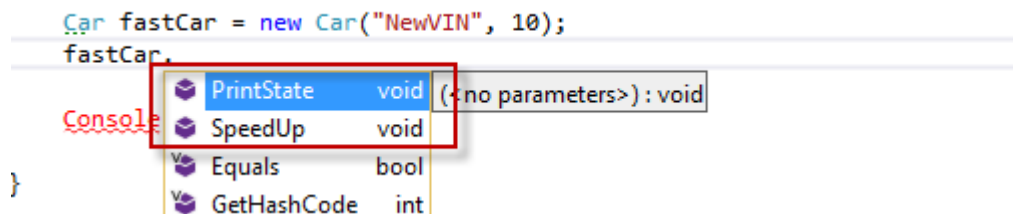


Рис. 27. Наследование методов базового класса

Обратите внимание, что экземпляр класса `fastCar` обладает всеми свойствами класса родителя, а именно, как видно из рис. 27 у класса `fastCar` появились методы `PrintState` и `SpeedUp`. В тоже время, как показано на рис. 28 есть собственные свойства.

```

Car fastCar = new Car("NewVIN", 10);
bool b = fastCar.isAirbagOk;

```

Рис. 28. Доступ к переменной-члену класса `Car`

В мире ООП существует и другая форма повторного использования кода: модель включения/делегации, также известная под названием отношение «имеет» (has-a) или агрегация, а также композиция. Эта форма повторного использования не применяется для установки отношений «родительский-дочерний». Вместо этого такое отношение позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность (когда необходимо) пользователю объекта. Давайте рассмотрим несколько примеров.

Ассоциация

Ассоциация – это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

```
class Team
{
}
class Player
{
    private Team myTeam;
    public void SetTeam(Team newTeam)
    {
        myTeam = newTeam;
    }
}
```

А в данном случае **Player** не является непосредственно наследником **Team**, но использует этот класс, чтобы обозначить принадлежность игрока к какой-то команде. Диаграмма классов приведена на рис. 29.

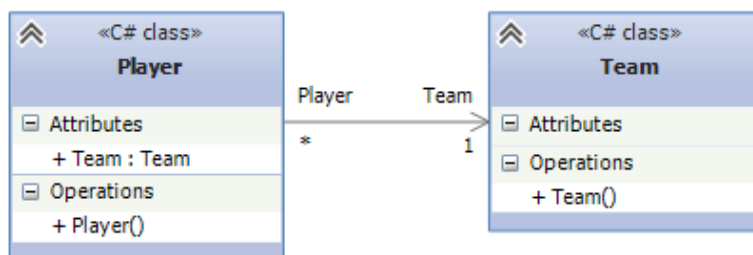


Рис. 29. Ассоциация

Композиция

Композиция определяет отношение HAS A, то есть отношение «имеет». Например, в класс автомобиля содержится объект класса электрического двигателя:

```
public class ElectricEngine
{ }

public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя — зависимой.

Еще раз посмотрите на предыдущий пример с игроком и командой. Обратите внимание, что мы передавали в метод **SetTeam** объект команды из внешнего окружения и не создавали новый экземпляр команды непосредственно в классе. Фактически, команда была создана где-то в программе, а потом уже готовый объект «пришел» в метод **SetTeam**. Очевидно, если игрок будет удален, то команда все равно будет существовать в системе. В примере с электродвигателем, напротив, объект двигателя создается непосредственно внутри класса **Car**. Это сильная связанность (что не является хорошим тоном) и теперь при удалении класса **Car** объект двигателя будет удален вместе с ним.

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик (рис. 30):

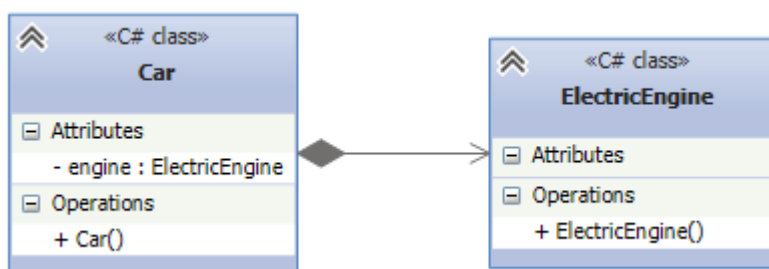


Рис. 30. Композиция

Агрегация

От композиции следует отличать агрегацию. Она также предполагает отношение (**has a**), но реализуется она иначе:

```

public abstract class Engine
{ }

public class Car
{
    Engine engine;
    public Car(Engine eng)
    {
        engine = eng;
    }
}
  
```

При агрегации реализуется слабая связь, то есть в данном случае объекты **Car** и **Engine** будут равноправны. В конструктор **Car** передается ссылка на уже имеющийся объект **Engine**. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Отношение агрегации на диаграммах UML отображается так же, как и отношение композиции, только теперь ромбик будет не закрашенным (рис. 31):

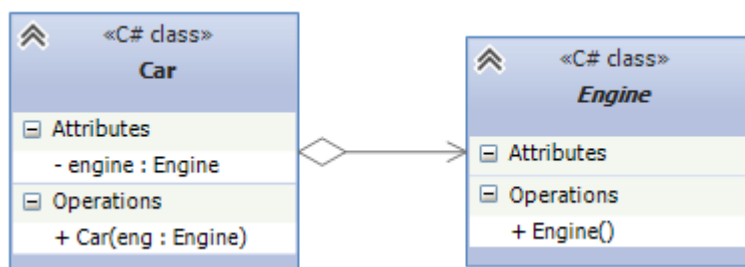


Рис. 31. Агрегация

Нельзя не указать несколько рекомендаций, которые весьма важны, но на данном этапе не стоит слишком серьёзно воспринимать написанное ниже, в данном случае это дано просто в ознакомительных целях.

Вместо композиции следует предпочитать агрегацию, как более гибкий способ связи компонентов. В то же время не всегда агрегация уместна. Например, у нас есть класс человека, который содержит объект нервной системы. Понятно, что в реальности, по крайней мере на текущий момент, невозможно вовне определить нервную систему и внедрить ее в человека. То есть в данном случае человек будет главным компонентом, а нервная система – зависимым, подчиненным, и их создание и жизненный цикл будет происходить совместно, поэтому здесь лучше выбрать композицию.

4.5.3. Роль полиморфизма

Прежде чем начать говорить о полиморфизме, хотелось бы обратить внимание, на то, что, как показывает практика тема полиморфизма становится наиболее непонятной или не очевидной для значительного числа начинающих разработчиков, столкнувшихся с объектно-ориентированным программированием, поэтому, если объяснение данного принципа, приведенное в методическом указании, будет не ясным, следует обратиться к сторонним источникам для получения более подробной информации.

Последний основной принцип ООП — полиморфизм. Он обозначает способность языка трактовать связанные объекты в сходной манере. В частности, этот принцип ООП позволяет базовому классу определять набор членов (формально называемый полиморфным интерфейсом), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с использованием любого количества виртуальных или абстрактных членов.

В сущности, виртуальный член — это член базового класса, определяющий стандартную реализацию, которая может быть изменена (или, говоря более формально, переопределена) в производном классе. В отличие от него, абстрактный метод — это член базового класса, который не предусматривает стандартной реализации, а предлагает только сигнатуру. Когда класс наследуется от базового класса, определяющего абстрактный метод, этот метод обязательно должен быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, они по существу переопределяют свою реакцию на один и тот же запрос.

Два абзаца, представленные выше, могут вызвать уйму вопросов, однако, на практике все не так сложно. Позвольте привести небольшой пример. Предположим, у нас есть объект «Человек», у которого заложено поведение перемещения. Если в метод «Перемещение» не передается никакого значения — считается, что человек идет, в противном случае — едет на чем-либо. Но как сделать так, чтобы в метод «Перемещение» могли передаваться объекты различных классов (машина, велосипед, автобус) ведь C# это строго типизированный язык. Многие уже могли догадаться, что нам на помощь приходит наследование и полиморфизм. Стоит сделать входным параметром тип Транспортное средство и теперь любой наследник (машина, велосипед и т.д) может быть передан в функцию перемещения. Принцип полиморфизма как бы дает нам гарантию (образует полиморфный интерфейс) того, что все объекты переданные в метод Перемещения у объекта Человек гарантировано могут ехать, потому что они наследники Транспортного средства. Давайте рассмотрим еще один пример, но уже с примером на языке C#.

Чтобы увидеть полиморфизм в действии представим некоторые детали иерархии фигур, показанной на рис. 26. Предположим, что в классе Shape определен виртуальный метод Draw (), не принимающий параметров.

```

class Shape
{
    public Shape() { }

    protected virtual void Draw()
    {
    }
}

```

Учитывая тот факт, что каждая фигура должна визуализировать себя уникальным образом, подклассы (такие как **Hexagon** и **Circle**) вольны переопределить этот метод по своему усмотрению (рис. 32).

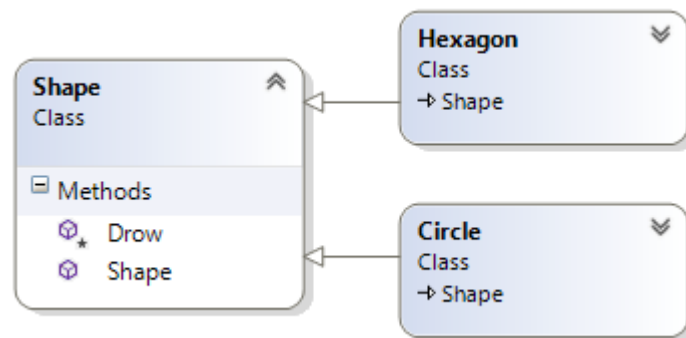


Рис. 32. Новая структура иерархии

В данном случае вызов **Draw()** на объекте **Circle** приведет к рисованию двумерного круга, а вызов **Draw()** на объекте **Hexagon** приведет к рисованию двумерного шестиугольника. Это достигается тем, что метод **Draw** – виртуальный. Т.е. наследники могут его переопределить (предоставить собственную реализацию), но даже если они этого не делают, он гарантированно у них будет в том виде, в котором он реализован у родителя. Мы опять вернулись к понятию полиморфного интерфейса. Родительский класс **Shape** дает гарантию, что все его наследники в том или ином виде будут уметь рисовать себя.

Рассмотрим следующий метод **Main()**, который заставляет массив типов-наследников **Shape** визуализировать себя с использованием метода **Draw()**:

```

static void Main(string[] args)
{
    Shape[] myShapes = new Shape[3];
    myShapes[0] = new Hexagon();
    myShapes[1] = new Circle();
    myShapes[2] = new Hexagon();
    foreach (Shape s in myShapes)
    {
        // Использовать полиморфный интерфейс!
        s.Draw();
    }
}

```

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

Код выше показывает, как на практике используется принцип полиморфизма. Массив `myShapes` содержит одновременно и объекты класса `Hexagon` и объекты класса `Circle`. Это доступно, так как сам массив типа `Shape` – базового класса. Но это весьма очевидно и не представляет интереса. Обратите внимание на цикл `foreach`, в котором `s` последовательно пробегает по всем объектам и вызывает корректный метод `Draw`, который реализован в соответствующем классе. Эта процедура доступна потому, что `Shape` гарантировал для всех своих классов наследников наличие метода `Draw` (обеспечил полиморфный интерфейс). В противном случае, пришлось бы создавать отдельный цикл для `Circle`, отдельный для `Hexagon` и так далее. Для переопределения метода используется конструкция `override` (пример: `public override void Draw() { }` в дочернем классе)

Понятие полиморфного интерфейса

Когда класс определен как абстрактный базовый (с помощью ключевого слова `abstract`), в нем может определяться любое количество абстрактных членов.

Абстрактные члены могут использоваться везде, где необходимо определить член, которые не предлагает стандартной реализации. За счет этого вы навязываете полиморфный интерфейс каждому наследнику, возлагая на них задачу реализации конкретных деталей абстрактных методов. Полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. На самом деле это интереснее, чем может показаться на первый взгляд, поскольку данная особенность ООП позволяет строить легко расширяемое и гибкое программное обеспечение.

Для иллюстрации реализуем (и слегка модифицируем) иерархию фигур. Подобно любому базовому классу, в `Shape` определен набор членов (в данном случае свойство `PetName` и метод `Draw()`), общих для всех наследников. Подобно иерархии классов сотрудников, нужно запретить непосредственное создание экземпляров `Shape`, поскольку этот тип представляет слишком абстрактную концепцию. Чтобы предотвратить прямое создание экземпляров `Shape`, можно определить его как абстрактный класс. Также, учитывая, что производные типы должны уникальным образом реагировать на вызов метода `Draw()`, давайте пометим его как `virtual` и определим стандартную реализацию.

```

abstract class Shape
{
    public Shape(string name = "NoName")
    { PetName = name; }
    public string PetName { get; set; }
    // Единственный виртуальный метод,
    public virtual void Draw()
    {
        Console.WriteLine("Inside Shape.Draw()");
    }
}

```

Обратите внимание, что виртуальный метод `Draw()` предоставляет стандартную реализацию, которая просто выводит на консоль сообщение, информирующее о том, что вызван метод `Draw()` базового класса `Shape`. Теперь вспомните, что когда метод помечен ключевым словом `virtual`, он предоставляет стандартную реализацию, которую автоматически наследуют все производные типы. Если дочерний класс решит, он может переопределить такой метод (используя конструкцию `override`), но он не обязан это делать. Учитывая это, рассмотрим следующую реализацию типов `Circle` и `Hexagon`:

```

class Circle : Shape
{
    public Circle() { }
    public Circle(string name) : base(name) { }
}
// Hexagon переопределяет Draw().
class Hexagon : Shape
{
    public Hexagon() { }
    public Hexagon(string name) : base(name) { }
    public override void Draw()
    {
        Console.WriteLine("Drawing {0} the Hexagon", PetName);
    }
}

```

Ясно, что это не особо интеллектуальное проектное решение для текущей иерархии. Чтобы заставить каждый класс переопределять метод `Draw()`, можно определить `Draw()` как абстрактный метод класса `Shape`, а это означает отсутствие какой-либо стандартной реализации. Для пометки метода как абстрактного в `C#` служит ключевое слово `abstract`. Не забывайте, что абстрактные методы не предоставляют вообще никакой реализации:

```

// Абстрактный базовый класс иерархии.
// Circle не переопределяет Draw().
abstract class Shape
{
    // Вынудить все дочерние классы определить свою визуализацию,
    public abstract void Draw();
}

```


Важно. Абстрактные методы могут определяться только в абстрактных классах. Попытка поступить иначе приводит к ошибке на этапе компиляции.

Методы, помеченные как **abstract**, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости). Здесь абстрактный класс **Shape** информирует типы-наследники о том, что у него имеется метод по имени **Draw ()**, который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться наследник.

Понятие интерфейсных типов

Реализацию полиморфного интерфейса можно обеспечить также на основе непосредственно объекта «Интерфейс».

Интерфейс представляет собой просто именованный набор абстрактных членов. Как упоминалось ранее, абстрактные методы являются чистым протоколом, поскольку они не предоставляют стандартной реализации. Специфичные члены, определяемые интерфейсом, зависят от того, какое точно поведение он моделирует. Другими словами, интерфейс выражает поведение, которое заданный класс или структура может избрать для поддержки. Более того, класс или структура может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений. Нетрудно догадаться, что в библиотеках базовых классов .NET поставляются сотни предопределенных интерфейсных типов, которые реализованы различными классами и структурами. Например, инфраструктура ADO.NET содержит множество поставщиков данных, которые позволяют взаимодействовать с определенной системой управления базами данных. Это означает, что в ADO.NET на выбор доступно несколько объектов подключения (**SqlConnection**, **OleDbConnection**, **OdbcConnection** и т.д.).

Несмотря на то, что каждый из этих объектов подключения имеет уникальное имя, определен в отдельном пространстве имен и (в некоторых случаях) упакован в отдельную сборку, все они реализуют общий интерфейс под названием **IDbConnection**.

После рассмотрения абстрактных классов интерфейс может показаться очень похожим на абстрактный базовый класс. Полиморфный интерфейс, устанавливаемый абстрактным родительским классом, обладает одним серьезным ограничением: члены, определенные абстрактным родительским классом, поддерживаются только производными типами. Тем не менее, в крупных программных системах очень часто раз-

работаются многочисленные иерархии классов, не имеющие общего родителя за исключением `System.Object`. Учитывая, что абстрактные члены в абстрактном базовом классе применимы только к производным типам, не существует никакого способа настройки типов в разных иерархиях на поддержку одного и того же полиморфного интерфейса.

Представленная выше информация может вызвать ряд вопросов и на начальном этапе к ней не стоит относиться слишком серьезно, она представлена только для ознакомления. Давайте перейдем от теории к практике и реализуем свой собственный интерфейс (рис. 33), добавим в него следующий код:

```
public interface IShape
{
    public void Draw();
}
```

Очевидно, что создание интерфейса лишь не значительно отличается от создания абстрактного класса. Теперь все наследники интерфейса `IShape` будут обязаны реализовать метод `Draw()`.

Информация. Перед названием интерфейсов рекомендуется ставить букву I, на сегодняшний день это стало стандартом де-факто. Пример: `IShape`, `ICore`, `IObject`

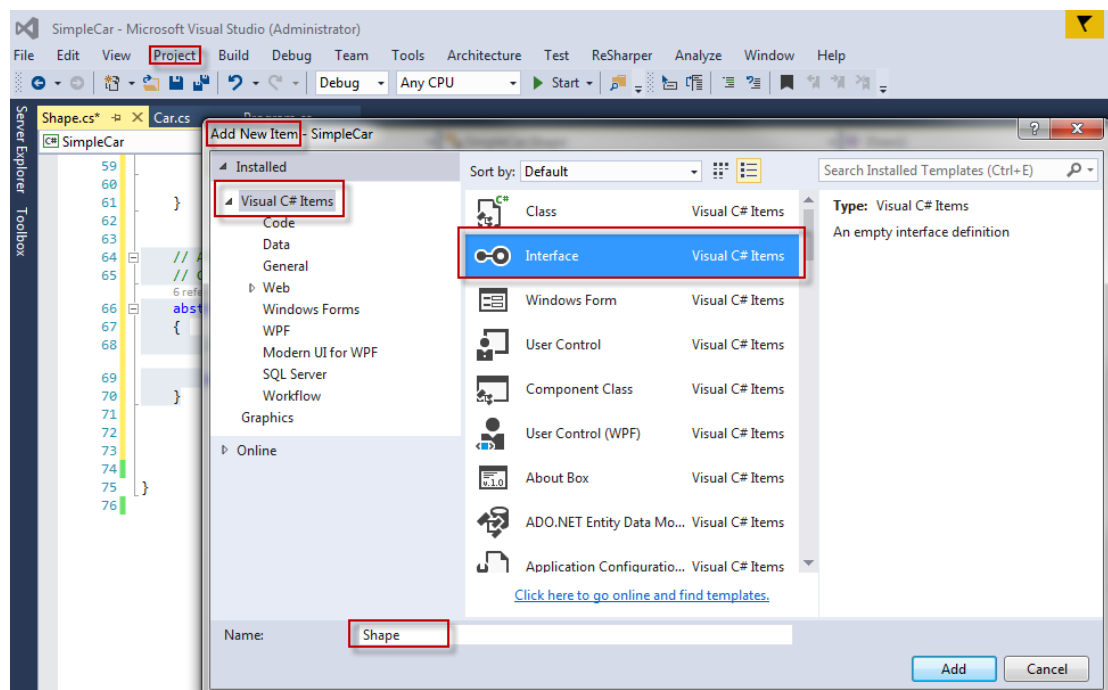


Рис. 33. Добавление нового интерфейса

В данных методических указаниях не будет подробного описания темы интерфейсов и их особенностей, проделайте данную работу самостоятельно, используя дополнительную литературу.

Инкапсуляция с использованием свойств .NET

В разделе инкапсуляция данных были рассмотрены подходы доступа к приватным переменным-членам класса, а именно подход с помощью открытых методов типа **Set**, **Get**. Например, **SetAge**, **GetAge**. Они позволяют скрыть приватный член **age** и предотвратить несанкционированный доступ к данному полю.

*Информация. Стоит запомнить, что метод типа **Set** всегда **void**, а метод **Get** всегда возвращает тип переменной-члена, которую он скрывает. Например **SetAge()** присваивает переменной **age** переданный в него аргумент, следовательно ничего не возвращает. В свое время **GetAge** должен вернуть внешнему окружению значений переменной **age**, следовательно имеет тип **int**. (типы функций обсуждались в предыдущем семестре).*

Вдобавок к возможности инкапсуляции полей данных с использованием традиционной пары методов **get/set**, в языках .NET имеется более предпочтительный способ инкапсуляции данных с помощью свойств. Прежде всего, имейте в виду, что свойства – это всего лишь упрощенное представление «реальных» методов доступа и изменения. Это значит, что разработчик класса по-прежнему может реализовать любую внутреннюю логику, которую нужно выполнить перед присваиванием значения (например, преобразовать в верхний регистр, очистить от недопустимых символов, проверить границы числовых значений и т.д.).

Ниже приведен класс **Employee**, который обеспечивает инкапсуляцию каждого поля с применением синтаксиса свойств вместо традиционных методов **get/set**.

```
class Employee
{
    // Поля данных,
    private string empName;
    private int empID;
    private float currPay;

    // Свойства.
    public string Name
    {
        get { return empName; }
        set
        {
            if (value.Length > 15)
```

```

        Console.WriteLine("Error! Name must be less than 16 charac-
ters!");
    }
    else
        empName = value;
    }
}
// Можно было бы добавить дополнительные бизнес-правила для установки
// этих свойств, но в данном примере в этом нет необходимости.
public int ID
{
    get { return empID; }
    set { empID = value; }
}
}

```

Обратите внимание на то, как методы `SetName`, `GetName` и так далее были заменены свойствами `Name {get; set;}`. Изучите представленный выше код самостоятельно и сделайте несколько экспериментов со свойствами и методами. В .NET часто используются автоматические свойства, они не будут представлены в данном методическом пособии и требуют самостоятельного изучения.

4.6. Заключение

Представленный в данном методическом указании материал отражает лишь незначительную часть всего разнообразия средств, которые представлены в .NET Framework для реализации принципов объектно-ориентированного программирования. Все темы, требующие самостоятельного изучения, указаны по мере изложения материала. Несмотря на сжатый объем, в данном пособии представлена информация, позволяющая получить базовые знания и навыки в области проектирования приложений с использованием объектно-ориентированного подхода.

На этом описание лабораторной работы № 3 завершено. Задание и требования к работе представлены в следующем подразделе.

4.7. Задание на лабораторную работу № 3

1. Модернизировать приложение из лабораторной работы № 2. Вынести функционал лабораторной работы № 2 в отдельные классы.

2. Преобразовать структуру для хранения данных в отдельные классы с собственным поведением и общим родительским классом. Реализовать базовый интерфейс (или абстрактный класс) и организовать структуру наследования.

3. Добавить, как минимум, две дополнительных переменных-члена и свойства реализованных классов.

4. Базовый класс должен содержать, как абстрактные методы, так и виртуальные. Все виртуальные методы должны быть переопределены в наследниках базового класса.

Подсказка: Структура классов должна представлять собой различные типы наследников (пример: напоминание, заметка для записной книжки или чартерный и обычный рейс для списка рейсов) реализующих свое поведение и базовый класс реализующий общее поведение. Для примера с записной книжкой три различных класса `class PersonalNote` и `class Reminder`, которые наследуются от базового класса `Record`.

5. Реализовать для каждого класса приватные переменные-члены и открытые свойства доступа. Все методы и переменные-члены должны иметь осмысленные названия, т.е. отображающие суть функции.

6. Реализовать и внедрить (в любом месте) блок программы, демонстрирующий принцип полиморфизма (параметр представлен типом базового класса, а на вход подаются объекты классов наследников).

7. Сформировать диаграмму классов средствами Visual Studio.

Требования к оформлению отчета

1. Формат листа – А4 (210x297 мм). Поля: слева – 2,5 см; справа – 1,5 см; сверху, снизу – 2 см.

2. Основной текст документа должен быть напечатан шрифтом Times New Roman Cyr, размером 12 пт. Межстрочный интервал – 1,5 строки, абзацный отступ – 1,25 см. Не добавлять разрыв между абзацами одного стиля. Выравнивание текста – по ширине.

3. Формат заголовков.

Абзац: отступ – 1,25 см, выравнивание – по левому краю.

Шрифт:

- заголовок первого уровня: размер – 12 пт, начертание – полужирный, регистр – все прописные;

- заголовок второго и последующих уровней: размер – 12 пт, начертание – полужирный, регистр – как в предложении.

4. Нумерация страниц производится автоматически.
5. Если позволяет структура документа, автоматически оформить оглавление.
6. Титульный лист должен быть оформлен по стандарту СТО ТПУ 2.5.01–2006 (<http://portal.tpu.ru/departments/head/methodic/standart>).
7. Рекомендуемая структура отчета:
 - титульный лист;
 - оглавление (если требуется);
 - введение;
 - ход работы (описывается последовательность выполнения заданий, при необходимости приводятся участки кода и скриншоты).
 - заключение.

Требования к защите

Лабораторная работа считается защищенной при наличии трех файлов:

1. Архив с исходным кодом.
2. Исполняемый файл.
3. Отчет по лабораторной работе.

5. ЛАБОРАТОРНАЯ РАБОТА № 4.

РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ И ТЕХНОЛОГИЯ СОМ

Выполнение данной лабораторной работы не требует пояснения вследствие наличия в .NET простой и прозрачной концепции работы с файловой системой. Ключевые задачи, которые ставятся перед слушателями курса в лабораторной работе № 4 – грамотное проектирование объектной модели для реализации заданного функционала. Все особенности проектирования структуры классов описаны в предыдущем разделе.

5.1. Задание на лабораторную работу № 4

Модернизировать приложение из лабораторной работы № 3.

1. Добавить возможность сохранять внесенные данные на жесткий диск компьютера. Реализовать возможность чтения и записи в файл.
2. Пользователю предлагается выбрать, в каком виде хранить данные:
 - а. хранить данные в бинарном виде;
 - б. хранить данные в виде XML;
 - с. хранить данные в текстовом формате (структура файла выбирается по желанию разработчика).
3. Для обеспечения работы с файловой системой реализовать отдельные классы. (Возможен вариант использования статических классов).
4. Добавить обработки ошибок: файл не открылся, нет прав доступа и т.д. Добавить обработку ошибок при введении неверных данных пользователем.
5. Добавить возможность работы с Microsoft Office:
 - а. для четных вариантов записать данных в Microsoft Excel;
 - б. для нечетных вариантов записать данных в Microsoft Word.

Подсказка. Обработку исключений реализовывать в виде конструкции try...catch

Требования к оформлению отчета

1. Формат листа – А4 (210х297 мм). Поля: слева – 2,5 см; справа – 1,5 см; сверху, снизу – 2 см.
2. Основной текст документа должен быть напечатан шрифтом Times New Roman Cyr, размером 12 пт. Межстрочный интервал – 1,5 строки, абзацный отступ – 1,25 см. Не добавлять разрыв между абзацами одного стиля. Выравнивание текста – по ширине.
3. Формат заголовков.
Абзац: отступ – 1,25 см, выравнивание – по левому краю.
Шрифт:
 - заголовок первого уровня: размер – 12 пт, начертание – полужирный, регистр – все прописные;
 - заголовок второго и последующих уровней: размер – 12 пт, начертание – полужирный, регистр – как в предложении.
4. Нумерация страниц производится автоматически.
5. Если позволяет структура документа, автоматически оформить оглавление.
6. Титульный лист должен быть оформлен по стандарту СТО ТПУ 2.5.01–2006 (<http://portal.tpu.ru/departments/head/methodic/standart>).
7. Рекомендуемая структура отчета:
 - титульный лист;
 - оглавление (если требуется);
 - введение;
 - ход работы (описывается последовательность выполнения заданий, при необходимости приводятся участки кода и скриншоты).
 - заключение.

Требования к защите

Лабораторная работа считается защищенной при наличии трех файлов:

1. Архив с исходным кодом.
2. Исполняемый файл.
3. Отчет по лабораторной работе.

Учебное издание

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Методические указания к выполнению лабораторных работ

Составитель

СКИРНЕВСКИЙ Игорь Петрович

Рецензент


*кандидат технических наук,
доцент кафедры АИКС ИК
М.С. Суходоев*

Компьютерная верстка *М.А. Зацепина*



Национальный исследовательский
Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета
сертифицирована в соответствии с требованиями ISO 9001:2008



ИЗДАТЕЛЬСТВО  **ТПУ**. 634050, г. Томск, пр. Ленина, 30.
Тел./факс: 8(3822)56-35-35, www.tpu.ru