МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ЛУГАНСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ»

В. В. Швыров
Д. А. Капустин
В. Н. Шишлакова

ПРОГРАММИРОВАНИЕ ДЛЯ ПЛАТФОРМЫ .NET

Часть 2. ПРИЛОЖЕНИЯ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Лабораторный практикум
для студентов 3 курса очной и заочной форм обучения
направления подготовки
44.03.04 Профессиональное обучение (по отраслям),
профиль: «Моделирование цифровых платформ»

Луганск
Издательство ЛГПУ
2024

УДК [004.04:004.514] (076.5)
ББК 32.973.202-018.2р3
Ш 35

Рецензенты:

Кривко Я. П. заведующий кафедрой высшей математики и методики преподавания математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный педагогический университет», доктор педагогических наук, доцент;

Мальцев Я. И. доцент кафедры прикладной математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный университет имени Владимира Даля», кандидат технических наук, доцент;

Давыскиба О. В. доцент кафедры фундаментальной математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный педагогический университет», кандидат педагогических наук, доцент.

Швыров В. В., Капустин Д. А., Шишлакова В. Н.
Ш35 Программирование для платформы .Net. Часть 2. Приложения с графическим интерфейсом: лабораторный практикум / В.В. Швыров, Д.А. Капустин, В.Н. Шишлакова; ФГБОУ ВО «ЛГПУ». – Луганск : Издательство ЛГПУ, 2024. – 116 с.

«Часть 2. Приложения с графическим интерфейсом» лабораторного практикума «Программирование для платформы .NET» содержит следующий блок из 13 лабораторных работ, которые посвящены изучению разработки современных приложений с графическим интерфейсом.

Предназначен для студентов 3 курса очной и заочной форм обучения направления подготовки 44.03.04 Профессиональное обучение (по отраслям), профиль: «Моделирование цифровых платформ».

УДК [004.04:004.514] (076.5)
ББК 32.973.202-018.2р3

Рекомендовано Учебно-методическим советом ФГБОУ ВО «ЛГПУ» в качестве лабораторного практикума для студентов очной и заочной форм обучения, обучающихся по направлению подготовки 44.03.04 Профессиональное обучение (по отраслям). Моделирование цифровых платформ (протокол № 10 от 21 мая 2024)

Содержание

Введение

Сегодня язык программирования C# считается одним из наиболее перспективных, быстроразвивающихся и востребованных в сфере ИТ. Если сравнивать его с другими популярными языками, C# является относительно молодым. Его первая версия увидела свет одновременно с релизом Microsoft Visual Studio .NET в феврале 2002 года. C# – это объектно-ориентированный язык программирования, который поддерживает такие концепции, как полиморфизм, наследование, переопределение операторов и статическая типизация. Объектно-ориентированный подход способствует созданию больших, но в то же время гибких, расширяемых и масштабируемых приложений. В настоящий момент C# активно развивается и с каждой новой версией добавляются новые интересные возможности, такие как lambda выражения, динамическое связывание, асинхронные методы и многое другое.

Данный лабораторный практикум предназначен для выполнения лабораторных работ по дисциплине «Программирование для платформы .Net» для студентов очной и заочной форм обучения по направлению подготовки 44.03.04 «Профессиональное обучение (по отраслям)». Профиль: «Моделирование цифровых платформ». В соответствии с учебным планом, по данной дисциплине предусмотрено 38 лабораторных работ. Во второй части лабораторного практикума представлен следующий блок из 13 лабораторных работ, которые посвящены изучению разработки современных приложений с графическим интерфейсом. В частности, рассматриваются возможности C# для работы с файлами и операционной системой, перечисления, структуры, разработка Windows Forms приложений и использование стандартных элементов управления. Кроме того, ряд лабораторных работ посвящен изучению таких важных аспектов синтаксиса C# как интерфейсы, делегаты, события.

В работах множество примеров иллюстрирует основные концепции объектно-ориентированной парадигмы программирования.

В структуре каждой лабораторной работы содержится блок, в котором представлены краткие теоретические сведения по теме работы, а также непосредственно задания для выполнения и контрольные вопросы.

Для успешной защиты лабораторной работы студенту необходимо детально изучить представленные теоретические сведения, выполнить все поставленные задачи, уметь ответить на контрольные вопросы. Кроме того, преподавателю необходимо предоставить все исходные файлы, которые содержат решения задач, а также сформировать отчет о выполненной работе.

Отчет по лабораторной работе должен содержать:

– титульную страницу, на которой указаны название ВУЗа, института, кафедры, тема лабораторной работы, курс, группа, ФИО студента, и другие сведения в соответствии с установленном образцом;

– основную часть, в которой приводится решение всех поставленных задач в виде последовательности скриншотов IDE, которые подписываются строкой комментарием, с кратким описанием действий;

– все листинги исходных кодов разработанных приложений;

– ответы на контрольные вопросы к лабораторной работе;

– выводы о том, что было сделано в работе.

Лабораторная работа № 14

Тема: **«Создание Windows Forms приложений. Использование стандартных элементов управления».**

Цель: изучить основы создания Windows Forms приложений с использованием языка программирования C#, а также освоить процесс проектирования и разработки интуитивно понятного и функционального пользовательского интерфейса с использованием стандартных компонент.

Краткие теоретические сведения

В Windows Forms форма – это визуальная поверхность, на которой выводится информация для пользователя. Обычно приложение Windows Forms строится путем помещения элементов управления на форму и написания кода для реагирования на действия пользователя, такие как щелчки мыши или нажатия клавиш. Элемент управления – это отдельный элемент пользовательского интерфейса, предназначенный для отображения или ввода данных.

При выполнении пользователем какого-либо действия с формой или одним из ее элементов управления создается событие. Приложение реагирует на эти события с помощью кода и обрабатывает события при их возникновении.

Windows Forms включает широкий набор элементов управления, которые можно добавлять на формы: текстовые поля, кнопки, раскрывающиеся списки, переключатели. Используя функцию перетаскивания конструктора Windows Forms в Visual Studio, можно легко создавать приложения Windows Forms. Достаточно выделить элемент управления курсором и поместить его в нужное место на форме.

Рассмотрим пример создания приложения HelloWorld. Также, добавим элементы управления на форму приложения и изучим некоторые свойства используемых элементов управления.

1. Добавление кнопки на форму

Необходимо выбрать Панель элементов, чтобы открыть всплывающее окно «Панель элементов» (рис.14.1).

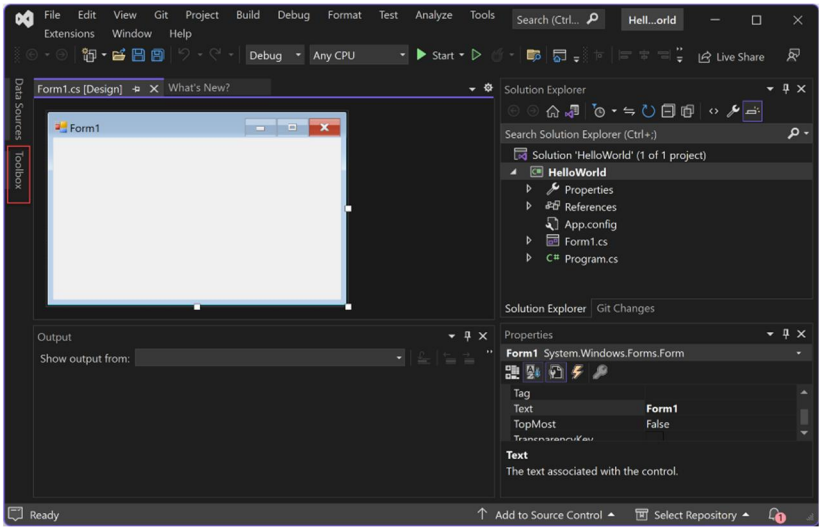


Рисунок 14.1. Окно редактора в режиме конструктора

Если Флажок всплывающего меню панели элементов не отображается, его можно открыть в строке меню. Для этого можно выбрать панель элементов. Или нажать клавиши CTRL+ALT+X.

Необходимо развернуть общие элементы управления и выберите значок «Закрепить», чтобы закрепить окно панели элементов (рис.14.2).

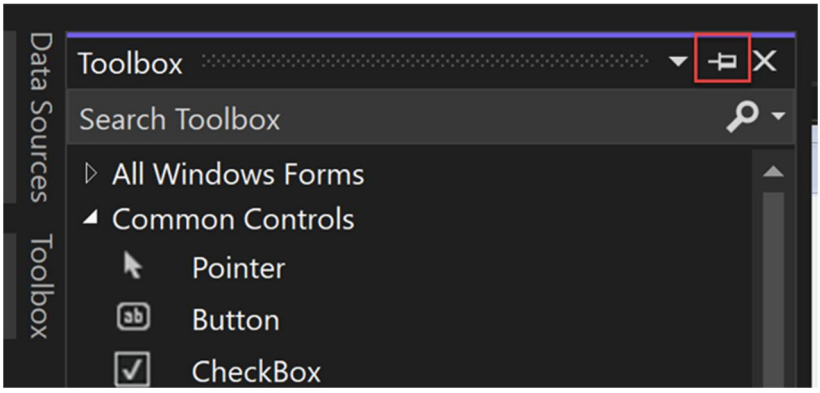


Рисунок 14.2. Закрепление панели элементов

Далее, нужно выбрать элемент управления Кнопка и перетащить его на форму(рис.14.3).

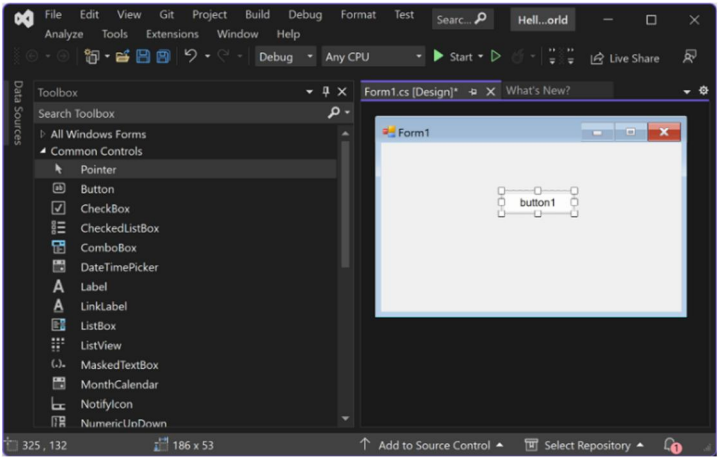


Рисунок 14.3. Добавление кнопки

В окне Свойства – свойство Text, позволяет изменять текст на кнопек (можно изменить button1 на «Click this») (рис.14.4).

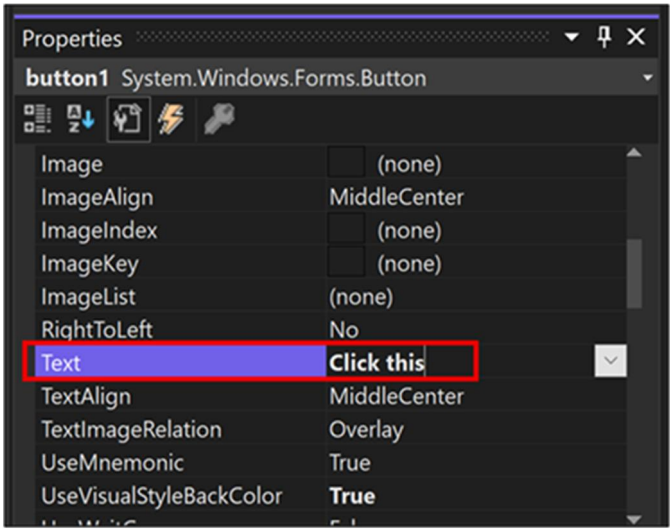


Рисунок 14.4. Настройка свойств

Если «Окно свойств» не отображается – его можно открыть в строке меню. Для этого нужно выбрать окно «Просмотр свойств». Или нажмите клавишу F4.

Например, в разделе Конструктор окна Свойства можно изменить имя с button1 на btnClickThis (рис.14.5).

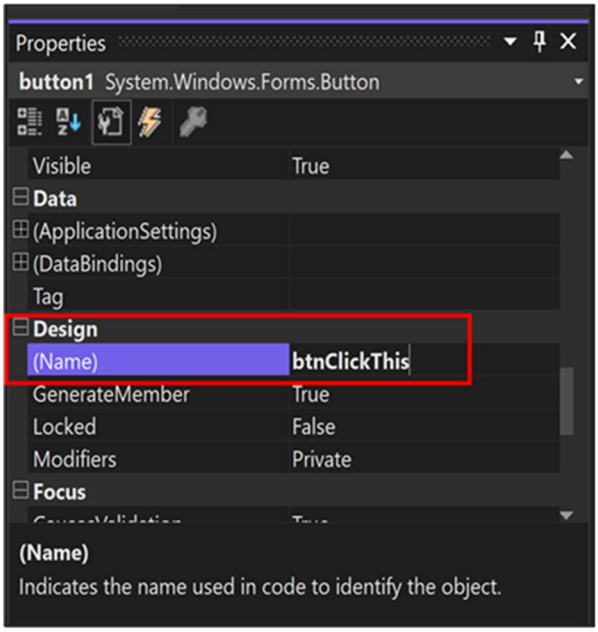


Рисунок 14.5. Редактирование имени

Если список в окне Свойства был упорядочен по алфавиту, button1 появится в разделе DataBindings.

2. Добавление метки на форму

После добавления элемента управления «Кнопка» (класс `Button`) для создания действия, добавим элемент управления «Метка» (класс `Label`), куда можно отправлять текст.

Элементы управления Label Windows Forms используются для вывода текста, который недоступен для изменения пользователем. Они используются для идентификации объектов в форме – например, чтобы предоставлять описание того, что будет

делать определенный элемент управления при его нажатии, или чтобы отображать сведения в ответ на событие времени выполнения или процесс в приложении. Так как элемент управления Label не может принимать фокус, его также можно использовать для создания клавиш доступа для других элементов управления.

Выберем элемент управления Метка в окне Панель элементов, а затем перетащим его на форму расположив под кнопкой.

В разделе Конструктор или Привязки данных окна Свойства изменим имя label1 на lblHelloWorld.

3. Добавление кода на форму

В окне Form1.cs [Конструктор] необходимо дважды щелкнуть по созданной кнопке, чтобы открыть окно Form1.cs.

(Кроме того, можно развернуть узел Form1.cs в обозревателе решений, а затем выбрать Form1.)

В окне Form1.cs после строки private void введем код

```
lblHelloWorld.Text = "Hello World!";
```

как показано на следующем снимке экрана (рис.14.6):

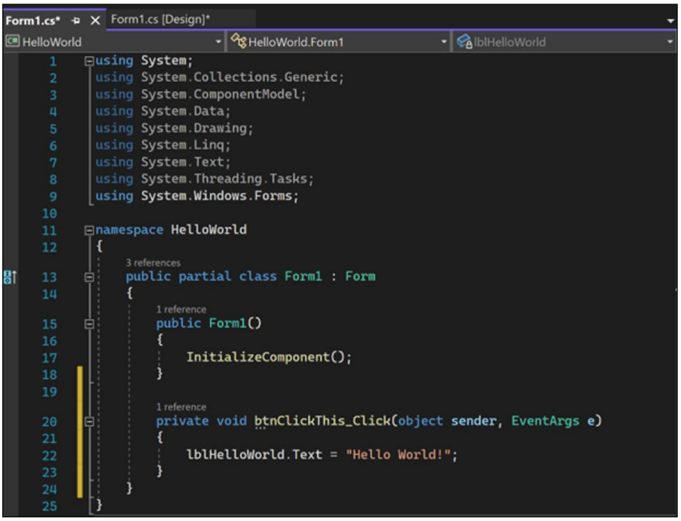


Рисунок 14.6. Окно редактора кода

4. Выполнение приложения

Для запуска приложения необходимо нажать кнопку Запустить (рис. 14.7).

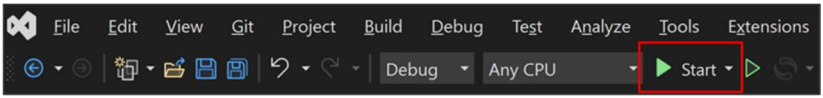


Рисунок 14.7. Запуск приложения

Будет выполнено несколько операций. В интегрированной среде разработки Visual Studio откроются окна Средства диагностики и Вывод. Кроме того, вне этой среды откроется диалоговое окно Form1.

Нажмите кнопку в диалоговом окне Form1. Обратите внимание, что текст label1 меняется на Hello World! (рис.14.8):

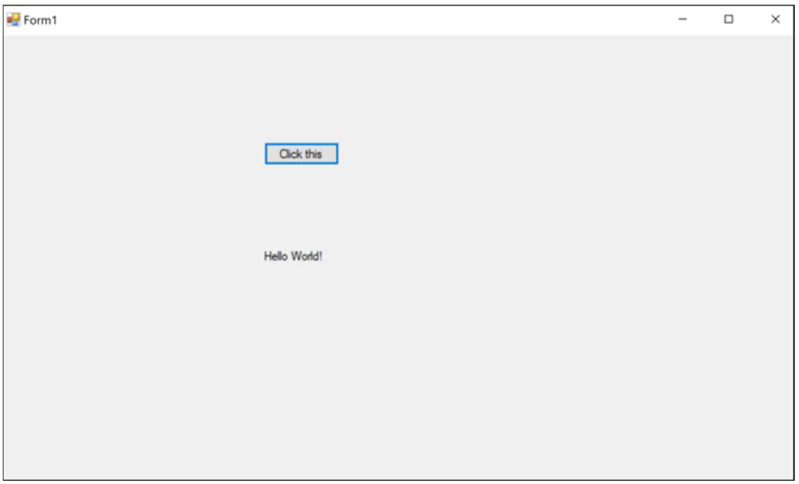


Рисунок 14.8. Окно формы

Закройте диалоговое окно Form1, чтобы завершить работу приложения.

Задания и порядок выполнения работы

Задание 1. Создайте Windows Forms приложение согласно примеру, приведенному в теоретическом материале. В приложение

необходимо добавить кнопку и метку. Нажатие кнопки должно менять текст в элементе label.

Задание 2. В созданном приложении измените текст названия формы на «Первое Windows Forms приложение. Автор - ФИО» (в качестве автора указать свои ФИО).

Задание 3. Изучите как выглядят на форме различные стандартные элементы управления и какие действия они выполняют. Опишите что делает каждый из элементов управления в отчете (рис.14.9).

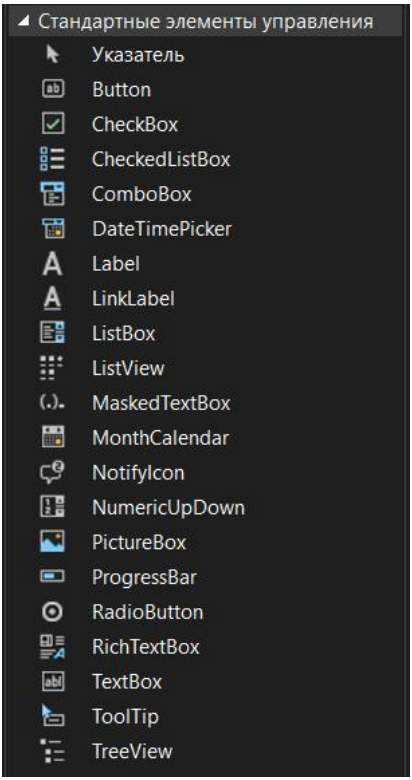


Рисунок 14.9. Стандартные элементы управления

Задание 4. Изучите как выглядят на форме элементы управления группы Контейнеры (рис. 14.8). Опишите, возможности данных элементов управления в отчете.

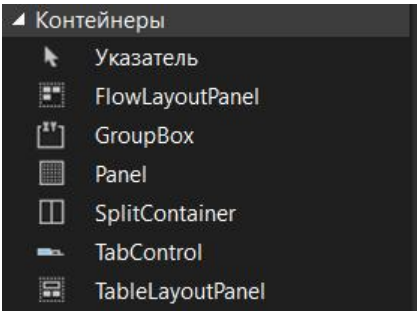


Рисунок 14.10. Группа элементов управления Контейнеры

Контрольные вопросы

1. Какие элементы управления предоставляет среда разработки для создания интерфейса в Windows Forms?
2. Как создать новый проект Windows Forms в Microsoft Visual Studio?
3. Какие основные этапы включает в себя разработка приложения в Windows Forms?
4. Какие свойства и методы доступны для элементов управления в Windows Forms и как их можно настроить?
5. Как добавить обработчик событий для элементов управления и какие основные события поддерживаются в Windows Forms приложениях?
6. Как организовать структуру проекта и расположить элементы управления на форме с использованием контейнеров и макетов?
7. Как работает событие MouseMove для элементов управления и как его можно использовать для создания интерактивного интерфейса?
8. Как использовать стандартные элементы для взаимодействия пользователя с приложением?
9. Для чего используются элементы управления группы Контейнеры?
10. Как изменить заголовок приложения?

Лабораторная работа № 15

Тема: **«Работа с файлами в C#».**

Цель: получить практические навыки работы с файловой системой в C#, изучить различные методы чтения и записи данных, а также разобраться в особенностях обработки ошибок и обеспечения безопасности при работе с файлами.

Краткие теоретические сведения

Для работы с файлами в пространстве имен `System.IO` предусмотрено два класса для работы с файлами – `File` и `FileInfo`. Рассмотрим, как можно использовать эти классы при работе с файловой системой в C#.

1. Класс File

Класс `File` рекомендуется использовать если вам необходима проверка безопасности используемых методов по работе с файлом, и вы не планируете использовать класс многократно. Основные методы работы с файлами в C# с использованием класса File следующие:

`Copy()` – копирует файл в новое место;

`Create()` – создает файл;

`Delete()` – удаляет файл;

`Move()` – перемещает файл в новое место;

`Exists(file)` – определяет, существует ли файл.

Например, для создания файла города.txt можно использовать код

```
static void Main(string[] args)
{
   File.Create("W:\\cslessons\\города.txt");
}
При работе с файлами все операции лучше всего размещать в блоки
try
try
{
    File.Create("WW:\\cslessons\\города.txt");
}
catch
{
```

```
    Console.WriteLine("Не могу создать файл, неверный путь к
файлу или нет доступа!");
}
```

Если бы была допущена ошибка в имени файла или в пути к нему, то в случае отсутствия обработки исключений и блока try в программе возникла бы ошибка (рис.15.1).

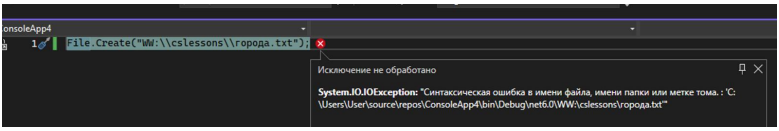


Рисунок 15.1 Синтаксическая ошибка в имени

Для записи в существующий файл можно использовать код

```
File. WriteAllText("W:\\cslessons\\города.txt", "текст");
```

Либо использовать добавление в конец файла

```
File.AppendAllText("W:\\cslessons\\города.txt",     "Донецк");
//допишет текст в конец файла
```

Метод Delete служит для удаления указанного файла

```
File.Delete("d:\\test.txt"); //удаление файла
```

2. Класс FileInfo

Ниже представлены наиболее часто используемые свойства и методы работы с файлами посредством класса FileInfo:

`CopyTo(path)` – копирует файл в новое место по указанному пути path

`Create()` – создает файл

`Delete()` – удаляет файл

`MoveTo(destFileName)` – перемещает файл в новое место

Свойство `Directory` – получает родительский каталог в виде объекта `DirectoryInfo`

Свойство `DirectoryName` – получает полный путь к родительскому каталогу

Свойство `Exists` – указывает, существует ли файл

Свойство `Length` – получает размер файла в байтах

Свойство `Extension` – получает расширение файла

Свойство `Name` – получает имя файла

Свойство `FullName` – получает полное имя файла

Для создания нового файла и записи в него данных можно воспользоваться методом Create():

```
using System;
using System.IO;
using System.Text;
namespace FileSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                FileInfo fileInfo = new FileInfo(@"c:\CSharp
Output\MyFile.txt");
                FileStream fs = fileInfo.Create();
                fs.Write(Encoding.UTF8.GetBytes("Привет,
мир!"));
                fs.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Или же, воспользоваться методом CreateText():

```
using System;
using System.IO;
namespace FileSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                FileInfo fileInfo = new FileInfo(@"c:\CSharp
Output\MyFile.txt");
                StreamWriter sw = fileInfo.CreateText();
                sw.WriteLine("Привет, мир!");
                sw.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

В отличие от Create() метод CreateText() возвращает объект типа StreamWriter, который предоставляет удобные методы и свойства для работы в том числе и со строками, поэтому во втором примере не понадобилось преобразовывать строку в массив байтов для записи в файл. В свою очередь, метод Create() удобно использовать для записи бинарных файлов.

Чтобы получить информацию о файле можно воспользоваться свойствами класса FileInfo, например, как показано ниже:

```
try
{
    FileInfo fileInfo = new FileInfo(@"c:\CSharp
Output\MyFile.txt");
    Console.WriteLine($"Полное имя файла:
{fileInfo.FullName}");
    Console.WriteLine($"Дата создания:
{fileInfo.CreationTime}");
    Console.WriteLine($"Дата последнего доступа:
{fileInfo.LastAccessTime}");
    Console.WriteLine($"Дата последней записи в файл:
{fileInfo.LastWriteTime}");
    Console.WriteLine($"Размер файла: {fileInfo.Length}");
    Console.WriteLine($"Имя файла: {fileInfo.Name}");
    Console.WriteLine($"Директория в которой содержится файл:
{fileInfo.DirectoryName}");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Для перемещения файла в другой каталог в C# используется метод MoveTo():

```
string path = @"c:\CSharp Output\MyFile.txt";
string newPath = @"c:\CSharp Output\SubDir\MyFile.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.MoveTo(newPath);
}
```

Перед тем, как перемещать файл в другой каталог необходимо обязательно проверить существование этого каталога

на диске и, если каталог отсутствует, то создать его, иначе возможно возникновение исключения.

Копирование файла в C# осуществляется с помощью метода CopyTo():

```
string path = @"c:\CSharp Output\MyFile.txt";
string newPath = @"c:\CSharp Output\SubDir\MyFile.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.CopyTo(newPath);
}
```

Методы CopyTo() и MoveTo() класса FileInfo также имеют переопределенные версии в которых вторым параметром выступает параметр булевого типа (bool) который указывает на то, необходимо ли перезаписать содержимое файла, если он уже содержится по указанному пути. Если этот параметр равен false (или используется метод с одним параметром), то при обнаружении по новому пути указанного файла C# сгенерирует исключение.

Для удаления файла с диска в C# используется метод Delete():

```
string path = @"c:\CSharp Output\MyFile.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.Delete();
```

3. Работа с дисками (класс DriveInfo)

Для представления информации о диске в пространстве имен System.IO используется класс DriveInfo. В этом классе находится статический метод GetDrives(), который возвращает информацию обо всех логических дисках компьютера. Также DriveInfo предоставляет ряд полезных свойств:

AvailableFreeSpace – указывает на объем доступного свободного места на диске в байтах доступного текущему пользователю

DriveFormat – получает имя файловой системы

DriveType – представляет тип диска

`IsReady` – готов ли диск (например, DVD-диск может быть не вставлен в дисковод)

`Name` – получает имя диска

`TotalFreeSpace` – получает общий объем свободного места на диске в байтах доступного всем пользователям

`TotalSize` – общий размер диска в байтах

`VolumeLabel` – получает или устанавливает метку тома

`RootDirectory` – возвращает информацию о корневом каталоге диска

Например, получим информацию обо всех дисках на компьютере:

```
using System;
using System.IO;
namespace FileSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] drives = DriveInfo.GetDrives();
            foreach (DriveInfo drive in drives)
            {
                Console.WriteLine($"Название: {drive.Name}");
                Console.WriteLine($"Тип: {drive.DriveType}");
                if (drive.IsReady)
                {
                    Console.WriteLine($"Объем диска:
{drive.TotalSize}");
                    Console.WriteLine($"Свободное
пространство: {drive.TotalFreeSpace}");
                    Console.WriteLine($"Метка:
{drive.VolumeLabel}");
                }
            }
        }
    }
}
```

4. Работа с каталогами (классы Directory и DirectoryInfo)

Для работы с каталогами в пространстве имен System.IO можно использовать два класса: `Directory` и `DirectoryInfo`. Во многом эти классы схожи по функциональности и оба класса

являются статическими, однако, следуя рекомендациям Microsoft, класс DirectoryInfo следует использовать в том случае, если в приложении подразумевается, что объект для работы с директориями будет использоваться многократно.

Класс Directory это статический класс, предоставляющий ряд статических методов для управления каталогами. Некоторые из этих методов:

CreateDirectory(path) – создает каталог по указанному пути path

Delete(path) – удаляет каталог по указанному пути path

Exists(path) – определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false

GetDirectories(path) – получает список каталогов в каталоге path

GetFiles(path) – получает список файлов в каталоге path

Move(sourceDirName, destDirName) – перемещает каталог

GetParent(path) – получение родительского каталога

GetDirectoryRoot(path) – возвращает для заданного пути сведения о томе и корневом каталоге по отдельности или сразу

GetCurrentDirectory() – получает текущий рабочий каталог приложения.

Получим список всех каталогов в каталоге C:\Windows:

```
foreach (string sub in Directory.GetDirectories(@"C:\Windows"))
    Console.WriteLine($"   {sub}");
```

Для получения каталога, из которого было запущено приложение можно воспользоваться методом Directory.GetCurrentDirectory(). Например:

```
string curDir = Directory.GetCurrentDirectory();
Console.WriteLine($"Программа запущена из каталога: {curDir}");
```

Класс DirectoryInfo предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Вот только некоторые из его свойств и методов:

Create() – создает каталог

CreateSubdirectory(path) – создает подкаталог по указанному пути path

Delete() – удаляет каталог

Свойство `Exists` – определяет, существует ли каталог

`GetDirectories()` – получает список каталогов

`GetFiles()` – получает список файлов

`MoveTo(destDirName)` – перемещает каталог

Свойство `Parent` – получение родительского каталога

Свойство `Root` – получение корневого каталога

Как можно видеть по описанию свойств и методов, часть того, что у класса `Directory` определено как метод (например, `Exists`) в `DirectoryInfo` определено как свойство, что позволяет использовать объект класса `DirectoryInfo` многократно, избегая возможно лишних проверок.

Для создания каталогов и подкаталогов в C# используются методы `DirectoryInfo.Create()`, `DirectoryInfo.CreateSubdirectory(path)`

Рассмотрим использование этих методов в следующем примере:

```
static void Main(string[] args)
{
    string root = @"C:\CSharp Output\";
    string subDir = @"FileSystem\bin";
    DirectoryInfo directory = new DirectoryInfo(root);
    if (!directory.Exists)
    {
        directory.Create();
    }
    DirectoryInfo                    newDir                    =
directory.CreateSubdirectory(subDir);
    Console.WriteLine(newDir.FullName);
}
```

Для вывода списка всех файлов в директории и их свойств можно использовать метод GetFiles

```
string pathToDirectory = @"w:\cslessons\";
// сохраняем названия всех файлов в переменную allFiles (массив
строк)
DirectoryInfo dI = new DirectoryInfo(pathToDirectory);
FileInfo[] allFiles = dI.GetFiles();
try
{
    // В цикле пройдём по всем файлам и проверим их дату
изменения
    foreach (FileInfo fi in allFiles)
```

```
    {               Console.WriteLine("{0}  |  {1}",  fi.Name,
fi.LastWriteTime.ToString());    }
}
catch (IOException ex)
{       Console.WriteLine("Произошла ошибка: {0}", ex.Message);
}
```

5. Использование потоков (классы FileStream, StreamReader, StreamWriter)

Класс FileStream имеет много перегруженных версий конструкторов, рассмотрим только некоторые из них, которые используются наиболее часто, а также параметры, используемые в этих конструкторах (см. табл. 15.1).

Наиболее распространенный способ создания FileStream:

```
public FileStream (string path, FileMode mode);
```

где path – относительный или абсолютный путь к файлу; mode – способ открытия файла. Здесь FileMode – это перечисление, имеющее следующие поля:

Таблица 15.1. Параметры конструктора FileStream

Имя	Описание
CreateNew	Создавать новый файл. Если файл уже существует, создается исключение IOException.
Create	Создавать новый файл. Если файл уже существует, он будет перезаписан. Если файл уже существует, но является скрытым, создается исключение UnauthorizedAccessException.
Open	Открыть существующий файл. Если файл не существует, то создается исключение FileNotFoundException
OpenOrCreate	Открыть файл, если он существует, в противном случае должен быть создан новый файл.
Truncate	Открыть и перезаписать существующий файл. Файл становится доступным только для записи. Попытка выполнить чтение из файла, открытого с помощью FileMode.Truncate, вызывает исключение ArgumentException.
Append	Открывает файл, если он существует, и находит конец файла, либо создает новый файл. Файл открывается только для записи.

Например:

```
FileStream file = new FileStream("d:\\test.txt", FileMode.Open
, FileAccess.Read); //открывает файл только на чтение
```

Задания и порядок выполнения работы

Задание 1. Написать приложение для получения и вывода информации обо всех дисках в системе.

Задание 2. Создать файл студенты.txt двумя способами. В файл добавить список из 5 студентов своей группы. Реализовать вывод информации из файла в консольное приложение.

Задание 3. Создать Windows Forms приложение. На форму добавить кнопку и элемент ListBox. Расположить файл студенты.txt в директории проекта. По нажатию кнопки в список на форме должны добавляться фамилии студентов из файла.

Задание 4. На форму добавить кнопку, которая в текущем каталоге приложения создает каталоги с именами, которые считываются из файла students.txt

Задание 5. На форму приложения добавить кнопку, которая сохраняет в файл info.txt информацию обо всех директориях, файлах и их свойствах, расположенных в директории проекта.

Контрольные вопросы

1. Какие классы в пространстве имен System.IO используются для работы с файлами и каталогами в C#?

2. Как открыть файл для чтения, записи или добавления в C#?

3. В чем разница между методами File.Open, File.Create и File.AppendAllLines для работы с текстовыми файлами?

4. Что такое StreamReader и StreamWriter, как их использовать для чтения и записи текстовых файлов?

5. В чем преимущество использования FileStream перед другими методами для чтения и записи двоичных данных?

6. Какие особенности нужно учитывать при работе с временными файлами в C#? Как их удалять после использования?

7. Что такое FileInfo класс и как его можно использовать для получения информации о файле или каталоге?

8. Как обработать ошибки, возникающие при работе с файлами, такие как FileNotFoundException, DirectoryNotFoundException и IOException?

9. Что такое многопоточное программирование и как оно влияет на работу с файлами в C#? Какие нужно использовать блокировки для обеспечения безопасности доступа к файлам в многопоточной среде?

10. Что такое буферизация и зачем она нужна при чтении и записи файлов в C#?

Лабораторная работа № 16

Тема: «Установка пакетов с помощью NuGet. Работа DOCX в C#».

Цель: изучить процесс установки дополнительных пакетов с помощью NuGet, а также освоить работу с форматом docx с помощью пакета DOCX.

Краткие теоретические сведения

Вместо создания своих собственных библиотек и определения своего собственного функционала можно использовать уже ранее созданные другими разработчиками библиотеки и добавить их свой проект. Для управления внешними библиотеками в виде отдельных пакетов Microsoft предоставляет специальный менеджер пакетов под названием Nuget.

Пакет NuGet можно установить в проекте Microsoft Visual Studio с помощью диспетчера пакетов NuGet, консоли диспетчера пакетов или .NET CLI. После установки пакета NuGet можно создать ссылку на него в коде с `using <namespace>` помощью инструкции, где <пространство имен> – это имя используемого пакета. После создания ссылки можно вызвать пакет через его API.

Физически NuGet-пакет представляет единый архивный файл с расширением .nupkg, который содержит скомпилированный код в виде библиотек dll и других файлов, используемых в коде. Также пакет включает некоторое описание в виде номера версии и вспомогательной информации.

Для упрощения работы с пакетами, их распространения Microsoft организовал глобальный репозиторий пакетов в виде сервиса nuget.org. И если разработчик хочет поделиться своими наработками, он может загрузить свой код в виде Nuget-пакета в этот репозиторий, а другие разработчики могут скачать этот пакет и использовать его в свой программе. Посмотрим, как устанавливать Nuget-пакеты и использовать их в своем проекте. Прежде всего стоит отметить, что есть различные способы установки пакетов. В данном случае рассмотрим установку пакетов с помощью .NET CLI и в Visual Studio.

1. Установка nuget-пакетов с помощью консоли диспетчера пакетов

Для примера возьмем такой популярный Nuget-пакет как Newtonsoft.Json, который предназначен для работы с кодом json. Для этого перейдем меню Средства, выберем Диспетчер пакетов NuGet и выберем Консоль диспетчера пакетом (рис. 16.1).

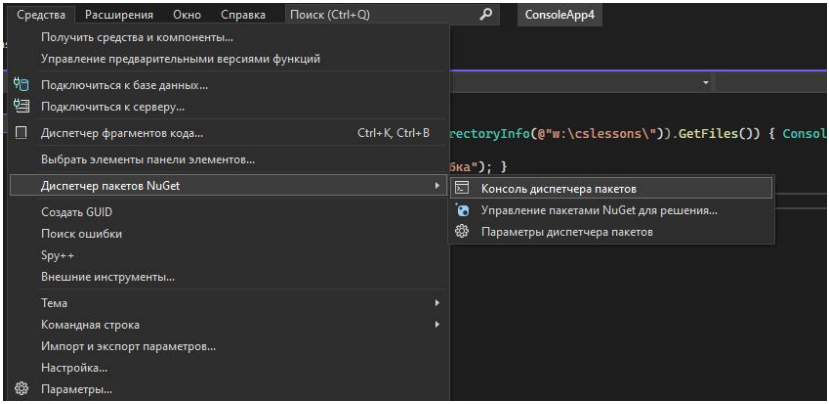


Рисунок 16.1. Консоль диспетчера пакетов

Затем, выполним команду для установки пакета `Install-Package Newtonsoft.Json\` (рис. 16.2).

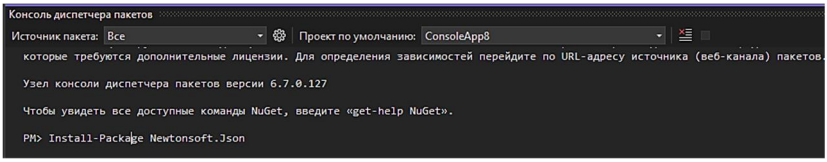


Рисунок 16.2. Установка пакета

2. Установка пакета в Visual Studio с помощью среды управления пакетами

Visual Studio обладает богатым функционалом в плане работы с nuget-пакетами. Так, возьмем также простейший проект консольного приложения и добавим в него тот же пакет Newtonsoft.Json. Для этого нажмем правой кнопкой на название проекта и в контекстном меню выберем Manage NuGet Packages.

Далее, в строке поиска на вкладке Обзор необходимо ввести название пакета, который необходимо установить, выделить найденный паке, выбрать требуемый проект для установки пакета и нажать кнопку Установить (рис. 16.3).

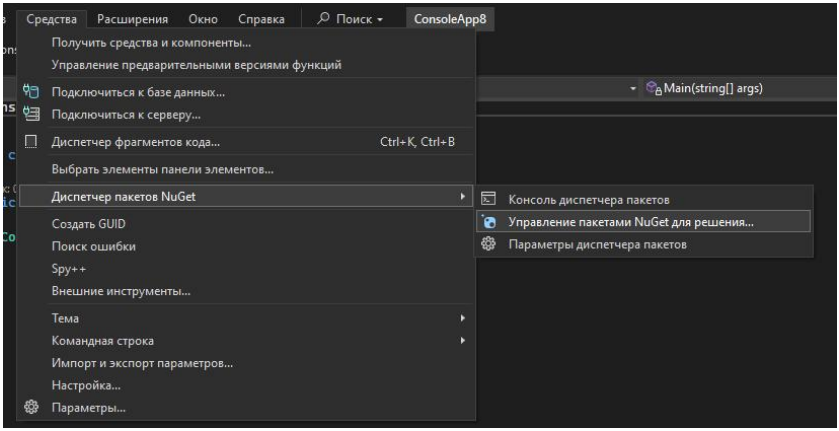


Рисунок 16.3. Установка пакета через диспетчер пакетов

Для примера, выполним снова установку пакета Newtonsoft.Json (рис. 16.4).

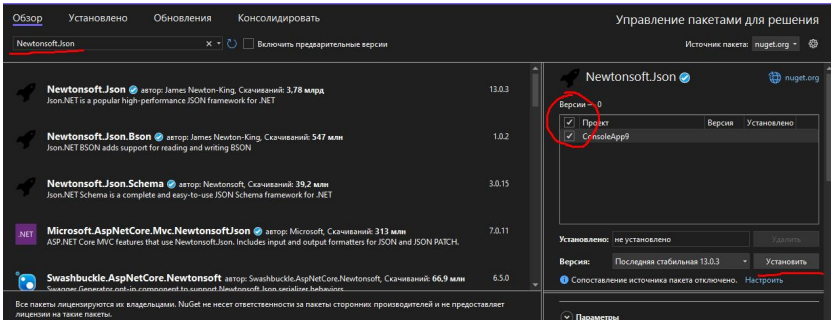


Рисунок 16.4. Установка пакета в текущий проект

Затем, необходимо подтвердить установку зависимостей пакета, если они есть (рис. 16.5).

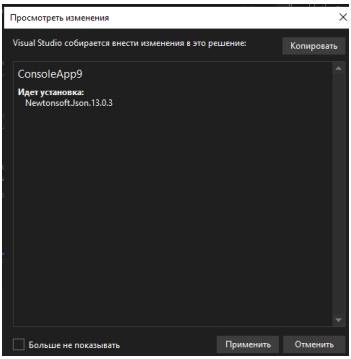


Рисунок 16.5. Конфигурация и установка зависимостей

После успешной установки – пакет должен отображаться на вкладке Установлено в списке установленных пакетов (рис 16.6)

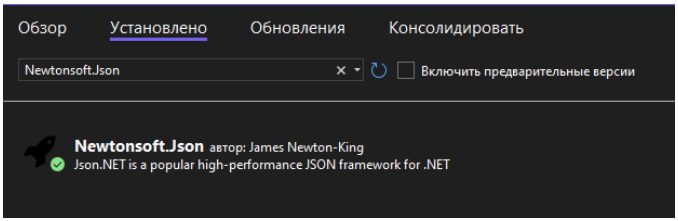


Рисунок 16.6. Просмотр установленных пакетов

Задания и порядок выполнения работы

Задание 1. Создать консольное приложение и установить пакет DOCX используя диспетчер пакетов (рис. 16.7).

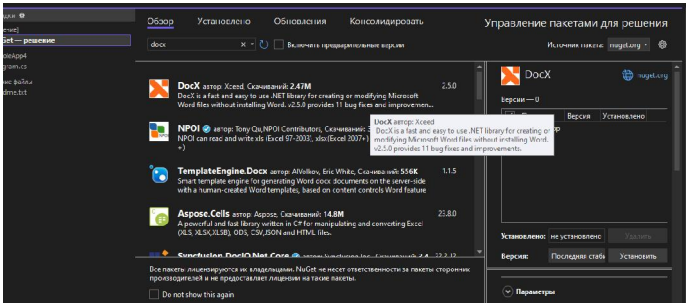


Рисунок 16.7. Установка пакета DOCX

Задание 2. Создать документ docx с текстом, содержащим свою ФИО, группу, курс. При создании документа использовать возможности пакета DOCX. (см. пример)

```
using Xceed.Words.NET;
string fileName = @"W:\cslessons\пример для Word.docx";
var doc = DocX.Create(fileName);
doc.InsertParagraph("Привет Word");
doc.Save();
```

Задание 3. Создать документ docx, который содержит таблицу со следующими столбцами ФИО, группа, курс, направление подготовки. Внести в таблицу свои данные. В качестве примера иллюстрирующего создание таблицы использовать следующий код

```
using Xceed.Document.NET;
using Xceed.Words.NET;
string fileName = @"W:\cslessons\пример для Word2.docx";
var doc = DocX.Create(fileName);
// создадим таблицу с 2 строками и 3  столбцами
Table t = doc.AddTable(2, 3);
t.Alignment = Alignment.center;
//заполнение значениями
t.Rows[0].Cells[0].Paragraphs.First().Append("Заголовок1");
t.Rows[0].Cells[1].Paragraphs.First().Append("Заголовок2");
t.Rows[0].Cells[2].Paragraphs.First().Append("Заголовок3");
t.Rows[1].Cells[0].Paragraphs.First().Append("текст 1");
t.Rows[1].Cells[1].Paragraphs.First().Append("текст 2");
t.Rows[1].Cells[2].Paragraphs.First().Append("текст 3");
doc.InsertTable(t);
doc.Save();
```

Задание 4. Создать документ docx, который содержит 5 параграфов со следующим форматированием: первый параграф – текст выровненный по центру, размер шрифта – 16пт, цвет – синий, шрифт – Arial, второй параграф – пустая строка, третий параграф – текст выделен курсивом, четвертый параграф – текст выделен жирным, пятый параграф – текст центрирован по левому краю. При выполнении задания использовать следующий пример кода

```
using System;
using System.Drawing;
using Xceed.Document.NET;
using Xceed.Words.NET;

```

```
namespace Word
{
    class Program
    {
        static void Main(string[] args)
        {
            // путь к документу
            string               pathDocument               =
AppDomain.CurrentDomain.BaseDirectory + "example.docx";

            // создаём документ
            DocX document = DocX.Create(pathDocument);

            // Вставляем параграф и указываем текст
            document.InsertParagraph("Тест");

            // вставляем параграф и передаём текст
            document.InsertParagraph("Тест").
                     // устанавливаем шрифт
                     Font("Calibri").
                     // устанавливаем размер шрифта
                     FontSize(36).
                     // устанавливаем цвет
                     Color(Color.Navy).
                     // делаем текст жирным
                     Bold().
                     // устанавливаем интервал между символами
                     Spacing(15).
                     // выравниваем текст по центру
                     Alignment = Alignment.center;

            // вставляем параграф и добавляем текст
            Paragraph paragraph = document.InsertParagraph();
            // выравниваем параграф по правой стороне
            paragraph.Alignment = Alignment.right;

            //   добавляем   отдельную   строку   со   своим
форматированием
            paragraph.AppendLine("Тест").
                     // устанавливаем размер шрифта
                     FontSize(20).
                     // добавляем курсив
```

```
                    Italic().
                    // устанавливаем точечное подчёркивание
                    UnderlineStyle(UnderlineStyle.dotted).
                    // устанавливаем цвет подчёркивания
                    UnderlineColor(Color.DarkOrange).
                    // добавляем выделение текста
                    Highlight(Highlight.yellow);
            // добавляем пустую строку
            paragraph.AppendLine();
            // добавляем ещё одну строку
            paragraph.AppendLine("Тест");

            // сохраняем документ
            document.Save();
        }
    }
}
```

Задание 5. Создать документ docx, который содержит параграф с гиперссылкой на сайт ЛГПУ. Использовать пример кода

```
using System;
using System.Drawing;
using Xceed.Document.NET;
using Xceed.Words.NET;

namespace Word
{
    class Program
    {
        static void Main(string[] args)
        {
            // путь к документу
            string                pathDocument                =
AppDomain.CurrentDomain.BaseDirectory + "example.docx";

            // создаём документ
            DocX document = DocX.Create(pathDocument);

            // создаём ссылку
            Hyperlink hyperlinkBlog =
                    document.AddHyperlink("progtask.ru",    new
Uri("https://progtask.ru"));

            // создаём параграф
```

```
            Paragraph paragraph = document.InsertParagraph();
            // центрируем содержимое
            paragraph.Alignment = Alignment.center;
            // добавляем текст
            paragraph.InsertText("Blog - ");
            // добавляем ссылку
            paragraph.AppendHyperlink(hyperlinkBlog)
                    // меняем цвет
                    .Color(Color.BlueViolet).
                    // устанавливаем вид подчёркивания

UnderlineStyle(UnderlineStyle.singleLine).
                    // устанавливаем размер шрифта
                    FontSize(16);

            // сохраняем документ
            document.Save();
        }
    }
}
```

Задание 6. Создать документ docx, который содержит произвольное изображение. Использовать код примера

```
using System;
using Xceed.Document.NET;
using Xceed.Words.NET;

namespace Word
{
    class Program
    {
        static void Main(string[] args)
        {
            // путь к документу
            string pathDocument =
AppDomain.CurrentDomain.BaseDirectory + "example.docx";
            string pathImage =
AppDomain.CurrentDomain.BaseDirectory + "image.jpg";

            // создаём документ
            DocX document = DocX.Create(pathDocument);

            // загрузка изображения
            Image image = document.AddImage(pathImage);

            // создание параграфа
            Paragraph paragraph = document.InsertParagraph();
```

```
            // вставка изображения в параграф
            paragraph.AppendPicture(image.CreatePicture());
            // выравнивание параграфа по центру
            paragraph.Alignment = Alignment.center;

            // сохраняем документ
            document.Save();
        }
    }
}
```

Задание 7. Создать документ docx, который содержит диаграммы различного вида. В качестве ряда подписей использовать значения 1, 2, 3, 4, 5. В качестве ряда значений – 5, 4, 3, 2, 1. В качестве примера использовать код

```
using System;
using System.Collections.Generic;
using Xceed.Document.NET;
using Xceed.Words.NET;
namespace Word
{
    class Program
    {
        static void Main(string[] args)
        {
            // путь к документу
            string pathDocument =
AppDomain.CurrentDomain.BaseDirectory + "example.docx";

            // создаём документ
            DocX document = DocX.Create(pathDocument);

            // добавляем линейную диаграмму
            document.InsertChart(CreateLineChart());
            // добавляем круговую диаграмму
            document.InsertChart(CreatePieChart());
            // добавляем столбцовую диаграмму
            document.InsertChart(CreateBarChart());
            // добавляем столбцовую диаграмму с 3d эффектом
            document.InsertChart(Create3DBarChart());

            // сохраняем документ
            document.Save();
        }
```

```
        private static Chart CreateLineChart()
        {
            // создаём линейную диаграмму
            LineChart lineChart = new LineChart();
            // добавляем легенду вниз диаграммы
            lineChart.AddLegend(ChartLegendPosition.Bottom,
false);

            // создаём набор данных и добавляем на диаграмму
            lineChart.AddSeries(TestData.GetSeriesFirst());
            // добавляем ещё один набор
            lineChart.AddSeries(TestData.GetSeriesSecond());

            return lineChart;
        }

        private static Chart CreatePieChart()
        {
            // создаём круговую диаграмму
            PieChart pieChart = new PieChart();
            // добавляем легенду слева от диаграммы
            pieChart.AddLegend(ChartLegendPosition.Left,
false);

            // создаём набор данных и добавляем на диаграмму
            pieChart.AddSeries(TestData.GetSeriesFirst());

            return pieChart;
        }

        private static Chart CreateBarChart()
        {
            // создаём столбцовую диаграмму
            BarChart barChart = new BarChart();
            // отображаем легенду сверху диаграммы
            barChart.AddLegend(ChartLegendPosition.Top,
false);

            // создаём набор данных и добавляем в диаграмму
            barChart.AddSeries(TestData.GetSeriesFirst());
            // создаём набор данных и добавляем в диаграмму
            barChart.AddSeries(TestData.GetSeriesSecond());

            return barChart;
```

```
        }

        private static Chart Create3DBarChart()
        {
            // создаём столбцовую диаграмму
            BarChart barChart = new BarChart();
            // отображаем легенду снизу диаграммы
            barChart.AddLegend(ChartLegendPosition.Bottom,
false);
            // добавление 3D эффекта
            barChart.View3D = true;

            // создаём набор данных и добавляем в диаграмму
            barChart.AddSeries(TestData.GetSeriesFirst());
            // создаём набор данных и добавляем в диаграмму
            barChart.AddSeries(TestData.GetSeriesSecond());

            return barChart;
        }
    }
    // класс с тестовыми данными
    class TestData
    {
        public string name { get; set; }
        public int value { get; set; }

        private static List<TestData> GetTestDataFirst()
        {
    List<TestData> testDataFirst = new List<TestData>();
    testDataFirst.Add(new TestData() { name = "1", value = 1
});
    testDataFirst.Add(new TestData() { name = "10", value = 10
});
    testDataFirst.Add(new TestData() { name = "5", value = 5
});
    testDataFirst.Add(new TestData() { name = "8", value = 8
});
    testDataFirst.Add(new TestData() { name = "5", value = 5
});
            return testDataFirst;
        }

        private static List<TestData> GetTestDataSecond()
        {
```

```
            List<TestData> testDataSecond = new
List<TestData>();
   testDataSecond.Add(new TestData() { name = "12", value = 12
});
   testDataSecond.Add(new TestData() { name = "3", value = 3
});
   testDataSecond.Add(new TestData() { name = "8", value = 8
});
   testDataSecond.Add(new TestData() { name = "15", value = 15
});
   testDataSecond.Add(new TestData() { name = "1", value = 1
});
            return testDataSecond;
        }

        public static Series GetSeriesFirst()
        {
            // создаём набор данных
 Series seriesFirst = new Series("First");
            // заполняем данными
 seriesFirst.Bind(TestData.GetTestDataFirst(), "name",
"value");
            return seriesFirst;
        }

        public static Series GetSeriesSecond()
        {
            // создаём набор данных
Series seriesSecond = new Series("Second");
            // заполняем данными
seriesSecond.Bind(TestData.GetTestDataSecond(), "name",
"value");
            return seriesSecond;
        }
    }
}
```

Контрольные вопросы

1. Что такое NuGet и для чего он используется?
2. Какие дополнительные пакеты NuGet могут быть полезны при разработке C# проектов?
3. Опишите процесс установки пакета NuGet в проект C#.
4. Какие способы установки пакетов через NuGet Вы знаеет?

5. Опишите основные возможности пакета DOCX и как можно работать с ним в C#?

6. Опишите команды для форматирования текста, которые используются в пакете DOCX.

7. Опишите основные классы и методы для работы с DOCX в пространстве имен Microsoft.Office.Interop.Word.

8. Каким образом выполняется работа с таблицами в пакете DOCX?

9. Какие проблемы могут возникнуть при использовании Microsoft.Office.Interop.Word и как их можно решить?

10. Опишите процесс создания диаграммы в документе docx с помощью пакета DOCX.

Лабораторная работа № 17

Тема: **«Абстрактные классы и интерфейсы в C#. Примеры работы с интерфейсами»**.

Цель: изучить принципы работы с абстрактными классами и интерфейсами в C#, научиться применять их для реализации полиморфизма и наследования, а также получить практические навыки их использования в объектно-ориентированном программировании.

Краткие теоретические сведения

Иногда бывает необходимо создавать классы, экземпляры которого нельзя создавать. Такие классы называются абстрактными. В каком случае нам может потребоваться абстрактный класс? Например, в случае, когда набор каких-либо сущностей относятся к одной и той же области знаний, но могут иметь различные реализации одних и тех же свойств и методов. Для примера, представим себе такую задачу – нам необходимо разработать систему учёта сотрудников в учебном заведении. Студент и преподаватель – это два человека у которых может быть имя, фамилия. При этом у преподавателя есть звание, должность, какие-то свои характеристики, например, стаж работы. У студента тоже могут быть свои, характерные только для него свойства, например, курс, на котором он учится, группа, средний балл по всем изучаемым дисциплинам и т.д. Таким образом, исходя из этих данных, можно выделить следующие важные для нас сущности – преподаватель и студент. Абстрактный класс в этом случае – человек, то есть этот класс будет чем-то общим между студентом и преподавателем.

1. Создание абстрактных классов

Создадим абстрактный класс, который будет представлять в программе какого-либо человека:

```
abstract class Person
{
    public string Name { get; set; }
    public string Family { get; set; }

    public Person(string name, string family)
```

```
    {
        Name = name;
        Family = family;
    }

    public string Display()
    {
        return $"{Family} {Name}";
    }
}
```

У класса определено два свойства: имя (`Name`) и фамилия (`Family`), а также конструктор и метод `Diasplay`, возвращающий строку, содержащую фамилию и имя человека. Ключевое слово `abstract` говорит нам о том, что класс является абстрактным и мы не можем воспользоваться конструктором абстрактного класса напрямую (создать объект абстрактного класса).
Теперь используем ключевую возможность ООП – наследование и создадим производный от `Person` класс студента.

```
class Student: Person
{
    string Group { get; set; }

    public Student(string name, string family, string group) :
base(name, family)
    {
        Group = group;
    }
}
```

Получается, определили свой конструктор для класса и добавили для класса свое свойство – `Group` (название группы, в которой учится студент). Теперь можно создать экземпляр этого класса и, например, воспользоваться методом `Diasplay`:

```
Person student = new Student("Вася", "Пупкин", "ГВН-105");
Student student1 = new Student("Ваня", "Иванов", "ГВН-105");
Console.WriteLine(student.Display());
Console.WriteLine(student1.Display());
```

Обратите внимание, что в первой строке объявили переменную `student` как `Person` (наш абстрактный класс), однако создали объект класса-потомка (`Student`) и по факту в нашей переменной лежит именно объект класса `Student`. Во втором случае использовали уже класс-потомок для описания типа

переменной. Теперь, в классе Student переопределим метод Display следующим образом:

```
public string Display(bool withGroup)
{
    if (withGroup)
        return string.Concat(new string[] { $"{Group} ",
Display() });
    else
        return Display();
}
```

То есть, в зависимости от параметра withGroup будем выводить либо просто имя и фамилию студента, либо перед именем и фамилией ставить группу. В коде ниже, оба вызова методов будут выполнены:

```
Console.WriteLine(student.Display()); //метод из абстрактного
класса
Console.WriteLine(student1.Display(true));
```

Соответственно, класс преподавателя можно сделать вот таким:

```
class Teacher: Person
{
    public int Seniority { get; set; }

    public Teacher(string name, string family, int seniority) :
base(name, family)
    {
        Seniority = seniority;
    }
}
```

Несмотря на то, что классы Student и Teacher – это вполне самостоятельные классы, описывающие разные сущности, в программе можем объявить их, используя абстрактный класс Person.

2. Интерфейсы в C#

До выхода C# 8.0, что такое интерфейс, с точки зрения программирования, можно было определить следующим образом – это именованная область кода, содержащая сигнатуры методов, свойств, индексаторов и событий. То есть, интерфейс, в его классическом понимании и представлении, не должен содержать никаких реализаций чего-либо – только сигнатуры.

Рассмотрим следующую задачу: нам необходимо наделить несколько классов одними и теми же методами. Пойдем следующим путем: если эти классы наши и классы родственны по своей сути, то можно выделить для них класс-предок, в котором реализовать необходимые нам методы и, соответственно, все наследники эти методы унаследуют и, при необходимости перегрузят или переопределят. Но как решать такую задачу, если: а) классы не родственны по своей сути; б) требуется, чтобы кодом (например, библиотекой классов) могли воспользоваться другие разработчики и написать свои реализации методов? И именно с этих двух моментов и начинается практическая значимость интерфейсов в C#.

3. Объявление интерфейса в C

Для того, чтобы объявить интерфейс в C# используется ключевое слово `interface`. В общем случае, объявление интерфейса выглядит следующим образом:

```
interface имя_интерфейса
{
    сигнатуры методов;
    сигнатуры свойств и т.д.
}
```

Имя интерфейса обычно начинается с заглавной буквы I. Например, вы можете встретить такие названия интерфейсов .NET как `IComparable`, `IEnumerable` и т.д. Такое именование сразу дает понять, что представлен интерфейс, а не класс. Технически, можно задать вообще любое имя интерфейса, не используя I в имени, но лучше придерживаться правил.

Для примера работы с интерфейсами в C# рассмотрим следующую ситуацию: есть человек, магазин и банк. Три неродственные сущности. И у человека, и у магазина, и у банка в распоряжении могут находиться деньги, например, у человека деньги хранятся в кошельке, у магазина – в кассе, у банка – в сейфе. Нам необходима сделать так, чтобы у всех трех классов были одинаковые методы работы с деньгами: метод, позволяющий положить деньги, например, в кошелек и метод, позволяющий изъять деньги из кошелька. Для этого объявим такой интерфейс:

```
interface IMoneyVault
{
    int MoneyAdd(int count);
    int MoneyRemove(int count);
}
```

Как можно видеть, интерфейс, во-первых, не содержит никаких реализаций методов – только сигнатуры и, во-вторых, методы интерфейса не имеют никаких модификаторов доступа – все они по умолчанию публичны и имеют модификатор `public` (по крайней мере до версии C# 8.0). После того, как был объявлен интерфейс, классы его (интерфейс) реализующие должны в обязательном порядке реализовать всё, что есть у интерфейса. В нашем случае – это два метода: `MonewAdd` и `MonewRemove`.

4. Реализация интерфейсов в C#

Чтобы реализовать интерфейс необходимо создать класс и указать у этого класса какой интерфейс (или интерфейсы) он реализуется. Например, создадим класс для описания человека:

Напишем следующие реализации методов:

```
public class Person : IMoneyVault
{
    private int currentMoney;
    public int MoneyAdd(int count)
    {
        return currentMoney += count;
    }
    public int MoneyRemove(int count)
    {
        return currentMoney -= count;
    }
}
```

Обратите внимание, что Visual Studio сразу же, как только записано имя реализуемого интерфейса, среда подчеркнет его, а в списке ошибок появятся одна или несколько ошибок следующего содержания:

```
Ошибка CS0535 «Person» не реализует член интерфейса
«IMoneyVault.MoneyAdd(int)».
```

Такие ошибки будут появляться до тех пор, пока не будет реализовано всё, что есть в интерфейсе.

После того, как интерфейс реализован, становится возможным использовать его методы. Например:

```
Person Person = new Person();//создаем объект
Console.WriteLine(Person.MoneyAdd(10)); //добавили деньги
Console.WriteLine(Person.MoneyRemove(5));//изъяли деньги
10
5
```

Аналогичным образом можно реализовать этот же интерфейс и в другом классе и написать для методов MoneyAdd и MoneyRemove. Например, путь у класса магазина при добавлении денег будет считаться сразу и сумма НДС:

```
public class Shop : IMoneyVault
{
    private int currentMoney;
    public int CurrentMoney { get=>currentMoney; }
    private double nalog;
    public double Nalog { get=>nalog; }
    public int MonewAdd(int count)
    {
        currentMoney += count;
        nalog += 0.13 * currentMoney;
        return currentMoney;
    }
    public int MonewRemove(int count)
    {
        currentMoney -= count;
        nalog -= 0.13 * currentMoney;
        return currentMoney;
    }
}
```

Проверим работу:

```
Shop Shop = new Shop();
Shop.MoneyAdd(1000);
Console.WriteLine($"Всего денег: {Shop.CurrentMoney}");
Console.WriteLine($"В том числе НДС: {Shop.Nalog}");
Всего денег: 1000
В том числе НДС: 130
```

Таким образом, сигнатуры методов и других сущностей, объявленных в интерфейсе, должны в точности совпадать. Например, такая реализация метода MoneyAdd ошибочна:

```
public double MonewAdd(int count)
 {
     currentMoney += count;
     nalog += 0.13 * currentMoney;
     return currentMoney;
 }
```

Так как в интерфейсе этот метод имеет другую сигнатуру – возвращаемое значение должно быть `int`, а не `double`.

Задания и порядок выполнения работы

Задание 1. Выполнить примеры в представленные в теоретической части материала.

Задание 2. Объявить интерфейс, который содержит описание методов для вычисления площади и периметра (см. пример кода ниже), также объявить несколько различных классов геометрических фигур, реализующих объявленный интерфейс: круг, трапеция, треугольник, прямоугольник, квадрат.

```
interface IGeometrical // объявление интерфейса
{
   void GetPerimeter();
   void GetArea ();
}
class Rectangle : IGeometrical //реализация интерфейса
{
   public void GetPerimeter()
   {
     Console.WriteLine("(a+b)*2");
   }

   public void GetArea()
   {
     Console.WriteLine("a*b");
   }
}
class Circle : IGeometrical //реализация интерфейса
{
   public void GetPerimeter()
   {
     Console.WriteLine("2*pi*r");
   }

   public void GetArea()
   {
     Console.WriteLine("pi*r^2");
   }
}

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

```
      List<IGeometrical> figures = new List<IGeometrical>();
      figures.Add(new Rectangle());
      figures.Add(new Circle());
      foreach (IGeometrical f in figures)
      {
        f.GetPerimeter();
        f.GetArea();
      }
      Console.ReadLine();
    }
}
```

Задание 3. Используя класс интерфейса и классы реализующие интерфейс разработать приложение для вычисления объема и площади шара, параллелепипеда, куба.

Контрольные вопросы

1. Дайте определение абстрактного класса в C#. Приведите пример использования.

2. Что такое интерфейс в C#? Чем он отличается от абстрактного класса? Приведите пример.

3. Что такое реализация интерфейса? Как ее создать в C#? Приведите пример.

4. Объясните принцип работы с интерфейсами в контексте наследования. Приведите пример кода.

5. Как реализовать множественное наследование с помощью интерфейсов в C#? Приведите код примера.

6. Что такое абстрактный интерфейс? Приведите примеры его использования.

7. Объясните, как работают виртуальные методы в абстрактных классах и реализациях интерфейсов. Приведите код примера.

8. Что такое частичная реализация интерфейса? Приведите пример ее использования.

9. В каких случаях используется ключевое слово `interface` вместо `abstract class`? Приведите примеры.

10. Объясните принцип композиции объектов с использованием интерфейсов в C#. Приведите код примера.

Лабораторная работа № 18

Тема: **«Перегрузка методов и операторов в пользовательских классах»**.

Цель: исследовать принципы перегрузки унарных и бинарных методов в пользовательских классах на примере разработки и тестирования конкретных программных реализаций, научиться применять различные виды перегрузки для повышения эффективности и гибкости разрабатываемого программного обеспечения.

Краткие теоретические сведения

В языке C# возможно объявить и создать в классе несколько методов с одним и тем же именем, но различающейся сигнатурой.

1. Сигнатура метода C

В C# сигнатура метода складывается из следующих элементов:

Имя метода

Количество и порядок параметров

Типы параметров

Модификаторы параметров

При этом при перегрузке метода его имя не меняется. Например, в классе Building создадим метод, который линейно увеличивает все три измерения здания (длину, ширину и высоту):

```
class Building
 {
     //код с описанием свойств не показан в целях сокращения
объема кода
          public void ExpandSize(int width, int length, int
height)
     {
         Width += width;
         Length += length;
         Height += height;
     }
 }
```

у этого метода будет следующая сигнатура:

```
 ExpandSize(int, int, int).
```

2. Перегрузка методов в C

Чтобы перегрузить метод, нужно оставить его имя, но изменить хотя бы одну из составляющих его сигнатуры. Например, приведенный выше метод принимает в качестве параметров только целочисленные значения. Возможно перегрузить метод, изменив тип данных входных параметров и получить два метода с одинаковым именем в одном классе:

```
public void ExpandSize(int width, int length, int height)
{
    Width += width;
    Length += length;
    Height += height;
}
public void ExpandSize(double width, double length, double
height)
{
    Width += width;
    Length += length;
    Height += height;
}
```

Также, можно поменять количество параметров:

```
//перегруженный метод
public void ExpandSize(double width, double length)
{
    Width += width;
    Length += length;
}
```

Теперь в классе имеется три метода изменения размеров здания, которые имеют следующие сигнатуры:

```
ExpandSize(int, int, int)
ExpandSize(double, double, double)
ExpandSize(double, double)
```

Перегруженные методы используются точно также, как и любые другие методы. При выборе того или иного метода Visual Studio подскажет есть ли у метода перегрузки и укажет их количество.

Чтобы воспользоваться одной из перегрузок метода, можно полностью написать имя метода и, поставить после названия метода круглую скобку.

3. Перегрузка арифметических операторов

Перегрузка операторов заключается в определении в классе, для объектов которого нужно определить оператор, специального метода:

```
public static возвращаемый_тип operator оператор(параметры)
{  }
```

Например, рассмотрим такой класс:

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

Этот класс определяет точку на плоскости с координатами (X, Y). Нам необходимо обеспечить векторное сложение и векторное вычитание. Компилятор C# умеет складывать, вычитать, сравнивать примитивные типы данных, однако про то, как сравнивать наши собственные классы и объекты он не знает. Технически, возможна запись:

```
Point point1 = new Point(10,10);
Point point2 = new Point(7,7);
Point point3 = new Point(point1.X+point2.X,point1.Y+point2.Y);
```

Однако, можно переопределить оператор сложения:

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
    //переопределенный оператор сложения
    public static Point operator +(Point p1, Point p2)
    {
        return new Point(p1.X + p2.X, p1.Y + p2.Y);
    }
}
```

Так как перегружаемый оператор будет использоваться для всех объектов данного класса, то он имеет модификаторы доступа public static. При сложении возвращается объект класса Point. Теперь можно сделать наш код более элегантным и понятным:

```
Point point1 = new Point(10, 10);
Point point2 = new Point(7, 7);
Point point3 = point1 + point2;//используем перегруженный оператор
Console.WriteLine($"X = {point3.X} Y={point3.Y}"); //X=17 Y=17
```

Аналогичным образом можно переопределять и другие арифметические операторы, в том числе операторы сложения, вычитания, умножения, деления и так далее. Например, вот так может выглядеть оператор * для выполнения операция скалярного умножения точки на плоскости:

```
public static Point operator *(double s, Point point)
{
    return new Point(s * point.X, s * point.Y);
}
```

И теперь можно умножать точку на любое число (выполнять скалярное умножение):

```
Point point2 = new Point(7, 7);
Point point3 = 2.5*point2;
Console.WriteLine($"X={point3.X}    Y={point3.Y}");    //X=17,5
Y=17,5
```

Также следует упомянуть, что операторы в C# бывают унарные и бинарные, но в любом случае один из параметров должен представлять тот тип – класс или структуру, в котором определяется оператор.

4. Перегрузка логических операторов

Немного иначе обстоит дело с перегрузкой логических операторов в C#. Отличие заключается в том, что операторы сравнения должны переопределяться попарно. Парными являются следующие операторы:

Операторы == и !=

Операторы < и >

Операторы <= и >=

Например, переопределим оператор >. Переопределенный оператор в классе Point может быть таким:

```
public static bool operator >(Point point1, Point point2)
{
    return (point1.X > point2.X) || ((point1.X == point2.X) &&
(point1.Y > point2.Y));
}
```

При этом, как только переопределим один из парных операторов, компилятор C# сообщит нам об ошибке:

```
Ошибка CS0216 Для оператора «Point.operator>(Point, Point)»
требуется, чтобы был определен соответствующий оператор «<«.
```

Поэтому, переопределяем и парный оператор <.

```
public static bool operator <(Point point1, Point point2)
{
    return (point1.X < point2.X) || ((point1.X == point2.X) &&
(point1.Y < point2.Y));
}
```

Можно также переопределить операторы true и false. Например, определим их в классе Point:

```
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
    public static bool operator true(Point p1)
    {
        return (p1.X != 0) && (p1.Y != 0);
    }
    public static bool operator false(Point p1)
    {
        return (p1.X == 0) && (p1.Y == 0);
    }
}
```

Использовать эти операторы можно следующим образом:

```
Point point1 = new Point(10, 10);
if (point1) //--используем операторы true/false у Point
    Console.WriteLine("Координаты точки point1 больше нуля");
else
    Console.WriteLine("Координаты точки point1 равны нулю");
Point point2 = new Point(0, 0);
```

```
if (point2) //--используем операторы true/false у Point
    Console.WriteLine("Координаты точки point2 больше нуля");
else
    Console.WriteLine("Координаты точки point2 равны нулю");
```

Консольный вывод будет следующим:

```
Координаты точки point1 больше нуля

Координаты точки point2 равны нулю
```

При переопределении операторов в C# следует учитывать следующее: при перегрузке не должны изменяться те объекты, которые передаются в оператор через параметры.

Это наиболее наглядно демонстрирует перегрузка унарных операторов, например, `++`. Например, можно определить для класса Point оператор инкремента:

```
public static Point operator ++(Point p1)
{
    p1.X += 1;
    p1.Y += 1;
    return p1;
}
```

Так как оператор `++` унарный, то он принимает один параметр – объект того класса, в котором данный оператор определен. Несмотря на то, что компилятор C# не предупредит нас об ошибке, это неправильное определение инкремента, так как оператор не должен менять значения своих параметров.

Более корректная перегрузка оператора инкремента будет выглядеть так:

```
public static Point operator ++(Point p1)
{
    return new Point(p1.X + 1, p1.Y + 1);
}
```

Использование унарного оператора будет таким же, как и при его использовании для простых типов данных. При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента (а также декремента), так как одна реализация будет работать в обоих случаях.

```
Point point1 = new Point(10, 10);
point1++;
++point1;
Console.WriteLine($"X = {point1.X} Y={point1.Y}"); //X=12 Y=12
```

Задания и порядок выполнения работы

Задание 1. Разработать класс “ComplexNumber”, который будет представлять комплексные числа, и перегрузить для него арифметические операции (+, -, *, /).

Задание 2. Реализовать класс “Matrix”, представляющий матрицы, и перегрузить операции сложения, вычитания, умножения матриц.

Задание 3. Разработать пользовательский класс “Vector”, представляющий векторы, и перегрузить методы для работы с векторами (сложение, вычитание, скалярное и векторное произведение).

Контрольные вопросы

1. Дайте определение понятий «унарный метод» и «бинарный метод» в контексте объектно-ориентированного программирования.
2. В чем заключается принцип перегрузки методов в объектно-ориентированных языках программирования?
3. Какие основные виды перегрузки методов существуют в языке C#?
4. Как осуществляется перегрузка методов с различным числом параметров в C#?
5. Как происходит разрешение конфликтов имен при перегрузке методов с одинаковым количеством и типами параметров в C#?
6. Как реализуется перегрузка методов с одинаковыми именами, но различными сигнатурами (типами возвращаемых значений и/или типами аргументов) в C#?
7. В каких ситуациях рекомендуется использовать перегрузку методов в пользовательских классах?
8. Какие преимущества дает использование перегрузки методов при разработке программного обеспечения на C#?
9. Опишите процесс тестирования и отладки кода с перегрузкой методов в C#.
10. Что такое «виртуальные методы» и как они связаны с перегрузкой?

Лабораторная работа № 19

Тема: **«Работа с регулярными выражениями в C#»**

Цель: изучить принципы работы с регулярными выражениями в языке программирования C#, научиться составлять и использовать регулярные выражения для обработки строк и текстовых данных в различных сценариях.

Краткие теоретические сведения

Регулярные выражения – это текстовые шаблоны, с помощью которых производится поиск и замена текста. Регулярные выражения, сами по себе – это набор специальных символов, который обрабатывается механизмом работы с регулярными выражениями. В C# используется свой механизм работы с регулярными выражениями, в Delphi – свой, в Python – третий и т.д.

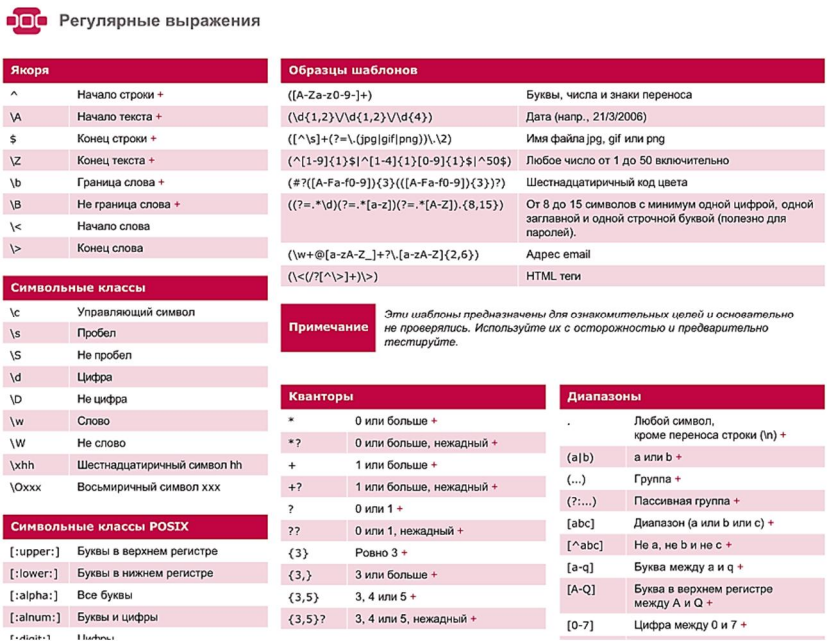
В C# есть достаточно много инструментов для работы с текстом (строками) – тот же класс StringBuilder или методы класса string. Представим такую задачу: передается большой фрагмент текста, например, web-страничка со статьей и необходимо перечислить все html-теги, которые встречаются в коде этой страницы. Или, скажем, извлечь с этой странички содержимое определенного тега. Можно пойти длинным путем – перечислить в программе все возможные теги, потом искать их позиции в тексте, запоминать, что нашли и т.д. Но и это не решит полностью нашу задачу. Однако, используя регулярные выражения, можно решить эту задачу буквально в несколько строк кода. Регулярные выражения, при должном навыке их использования, превращаются в по-настоящему мощнейший механизм обработки текста.

1. Основы построения регулярных выражений

Прежде, чем переходить непосредственно к классам C# для работы с регулярными выражениями, нам необходимо – понять суть и принцип их работы. Попробуем онлайн-сервис для работы с регулярными выражениями https://regex101.com/. Можно

сравнить различные версии регулярных выражений (для Python, C# и т.д.).

Любое регулярное выражение – это специальный шаблон, по которому будет происходить поиск текста. В C# выделяют несколько групп символов и их последовательностей из которых может состоять регулярное выражение: escape-символы, категории знаком, привязки и т.д. (рис. 19.1).



Регулярные выражения

Якоря	
^	Начало строки +
\A	Начало текста +
$	Конец строки +
\Z	Конец текста +
\b	Граница слова +
\B	Не граница слова +
\<	Начало слова
\>	Конец слова

Символьные классы	
\c	Управляющий символ
\s	Пробел
\S	Не пробел
\d	Цифра
\D	Не цифра
\w	Слово
\W	Не слово
\xhh	Шестнадцатиричный символ hh
\Oxxx	Восьмиричный символ xxx

Символьные классы POSIX	
[:upper:]	Буквы в верхнем регистре
[:lower:]	Буквы в нижнем регистре
[:alpha:]	Все буквы
[:alnum:]	Буквы и цифры

Образцы шаблонов	
([A-Za-z0-9-]+)	Буквы, числа и знаки переноса
(\d{1,2}\/\d{1,2}\/\d{4})	Дата (напр., 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	Имя файла jpg, gif или png
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Любое число от 1 до 50 включительно
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Шестнадцатиричный код цвета
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	От 8 до 15 символов с минимум одной цифрой, одной заглавной и одной строчной буквой (полезно для паролей).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Адрес email
(\<(/?[^\>]+)\>)	HTML теги

Примечание *Эти шаблоны предназначены для ознакомительных целей и основательно не проверялись. Используйте их с осторожностью и предварительно тестируйте.*

Кванторы	
*	0 или больше +
*?	0 или больше, нежадный +
+	1 или больше +
+?	1 или больше, нежадный +
?	0 или 1 +
??	0 или 1, нежадный +
{3}	Ровно 3 +
{3,}	3 или больше +
{3,5}	3, 4 или 5 +
{3,5}?	3, 4 или 5, нежадный +

Диапазоны	
.	Любой символ, кроме переноса строки (\n) +
(a|b)	a или b +
(...)	Группа +
(?:...)	Пассивная группа +
[abc]	Диапазон (a или b или c) +
[^abc]	Не a, не b и не c +
[a-q]	Буква между a и q +
[A-Q]	Буква в верхнем регистре между A и Q +
[0-7]	Цифра между 0 и 7 +

Рисунок 19.1. Шаблоны регулярных выражений

Итак, попробуем воспользоваться возможностями регулярных выражений, написав первое регулярное выражение – найдем все HTML-теги на странице. Регулярное выражение будет иметь вид:

```
(\<(/?[^\>]+)\>)
```

Переходим на сайт https://regex101.com/, выбираем слева в меню .NET (C#) и вставляем регулярное выражение в строку сверху (рис. 19.2):

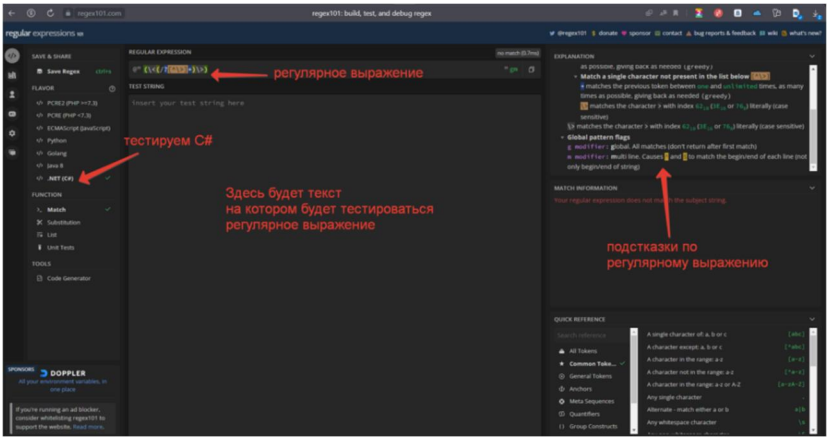


Рисунок 19.2. Поиск выражений по шаблону

Преимущество сайта https://regex101.com/ в том, что помимо непосредственно теста регулярного выражения, также можно, при необходимости, изучить из чего состоит регулярное выражение. Теперь возьмем любой html-код и вставим его в поле текста. В итоге получим следующий результат (рис. 19.3):

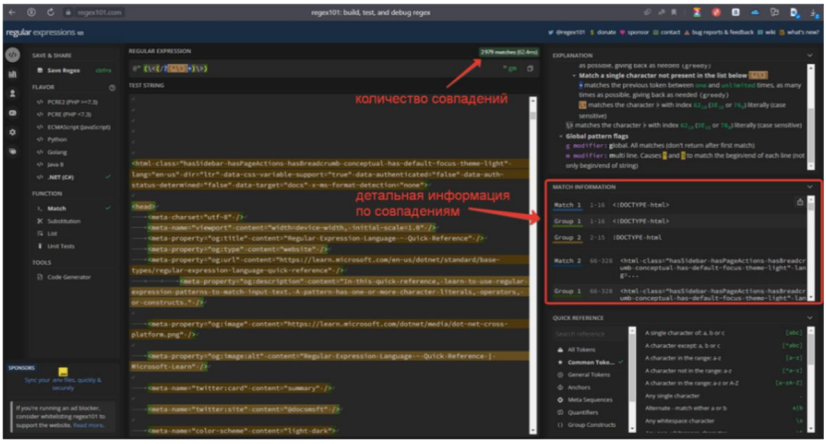


Рисунок 19.3. Результаты поиска

За 62 миллисекунды получили 2979 совпадений (matches) по каждому совпадению была выведена подробная информация. Рассмотрим детально строение регулярого выражения:

Первые круглые скобки (...)в регулярных выражениях так описывается группа (Group). Группа может состоять из набора символов. В нашем случае, указываем шаблону, что в эту группу попадает всё, что будет соответствовать шаблону, т.е. html-тег целиком, например, или и т.д.

Последовательность \<в нашем случае это означает, что следующий за косой чертой символ будет читаться буквально – означать ровно то, что он и означает, то есть начало HTML-тега <.

Вторые круглые скобки (...) – вторая группа, в которую попадут все символы, исключая треугольные скобки <>

Последовательность /? говорит о том, что символ / должен встретиться 0 или 1 раз. Символ ? в регулярных выражениях ещё называют квантификатором или квантором. Квантификатор указывает сколько раз должен встретиться стоящий перед ним символ или группа символов (токен).

Последовательность в квадратных скобках [^\>] представляет собой диапазон. В нашем случае дословно обозначает, что в проверяемом диапазоне не должно быть (^) символов \ и >. Все остальные символы могут встречаться.

Символ + говорит, что диапазон может встретиться 1 или более раз.

Последовательность \> указывает на то, что сравниваемый текст должен закончиться символом >.

Собственно, следуя этому описанию, мы и получили результаты, а именно – 2979 совпадений (Matches) где каждое совпадение содержит по две группы (Group) текста% в первой группе – весь тег целиком, во втором внутренность тега, включая символ /. Попробуем изменить регулярное выражение следующим образом:

```
(\</.?\>)
```

Вместо диапазона символов указали, что после последовательности символов </ может встретиться любой символ (за это отвечает символ точки – .) 0 или 1 раз (?) и, затем, должны встретить закрывающую скобку тега > В результате обработки текста с использованием этого шаблона получим все закрывающие

html-теги, состоящие из одного символа, например, , </p> и т.д.

Рассмотрим другой пример. Что означает такое регулярное выражение:

```
\s*(\w*н{2}\w*)\s*
```

Последовательность \s говорит о том, что ищем пробел. символ * говорит о том, что пробел (предыдущий токен) должен встретиться один или более раз

Далее начинается группа (...):

– в группе должны содержаться любые алфавитно-цифровые символы (\w) один или более раз (*);

– последовательность н{2} читается как «символ н должен встретиться строго два раза» (квантификатор {2});

– затем в группе должны встретиться снова любые символы, кроме пробела любое количество раз (w*);

– и шаблон заканчивается одним или несколькими пробелами \s*.

Таким образом, регулярное выражение должно найти в тексте все слова, в которых встречается двойная н. При этом, у регулярного выражения есть недостаток – в совпадение могут попасть лишние пробелы. Например, если перед словом поставить 100 пробелов, то все они попадут в совпадение (Match), но, при этом, группа будет содержать только слово.

Рассмотрим какие инструменты нам предлагает C# для работы с регулярными выражениями.

2. System.Text.RegularExpressions

Инструменты для работы с регулярными выражениями в C# содержатся в пространстве имен System.Text.RegularExpressions. Основным классом для работы с регулярными выражениями является класс Regex. Этот класс содержит как статические методы, так и методы, которые можно использовать только после создания объекта типа Regex.

Для примера, используем один из ключевых методов класса Regex – IsMatch, который возвращает true/false в зависимости от того, найдены ли совпадения в тексте или нет:

```
string pattern = @"\s*(\w*н{2}\w*)\s*"; //задаем регулярное
выражение
if (Regex.IsMatch("Анна наполнила ванну", pattern))
Console.WriteLine("Найдены совпадения"); //совпадения будут
найдены 100%
else
Console.WriteLine("Совпадения не найдены");
```

Здесь используется статический метод `IsMatch` в который первым параметром передаем обрабатываемый текст, а вторым – регулярное выражение. Стоит обратить внимание на то, как записывается регулярное выражение `(pattern)` в `C#`. Нами использовался символ `@`, указавающий, что строку надо читать буквально, чтобы символы после `\` не читались как `escape`-последовательности и не получить в итоге ошибки. Второй вариант записи менее читабельный – использовать двойную косую черту `\\`. То есть записывать регулярное выражение вот так:

```
"\\s*(\\w*н{2}\\w*)\\s*"
```

3. Настройки работы `Regex` (перечисление `RegexOptions`)

У `Regex` определено несколько конструкторов, позволяющих провести начальную инициализацию свойств объекта. В качестве параметров, можно передать в конструктор регулярное выражение, а также настройки в виде перечисления `RegexOptions`, которое состоит из следующих основных значений:

`Compiled` – компилировать регулярное выражение в сборку. Эта настройка позволяет ускорить работу регулярного выражения, особенно, если предполагается обработка больших объемов текста.

`CultureInvariant` – игнорировать настройки культуры.

`IgnoreCase` – игнорировать регистр символов в тексте.

`IgnorePatternWhitespace` – удаляет из регулярного выражения все пробелы и разрешает использовать комментарии, начинающиеся с символа #.

`Multiline` – текст необходимо рассматривать в многострочном режиме. При таком режиме символы `^` и `$` обозначают, соответственно, начало и конец строки, а не начало и конец всего текста.

`RightToLeft` – текст необходимо считывать справа налево

`Singleline` – текст необходимо рассматривать как одну строку. В этом случае символ `.` соответствует любому символу, в том числе последовательности `\n` (переход на новую строку)

Методы `Regex` и поиск последовательностей. Ниже представлены основные методы класса `Regex`. Каждый из этих методов также имеет ряд перегруженных версий.

`IsMatch` – указывает, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению.

`Match` – ищет в указанной входной строке первое вхождение регулярного выражения

`Matches` – ищет в указанной входной строке все вхождения регулярного выражения.

`Replace` – в указанной входной строке заменяет все строки, соответствующие указанному регулярному выражению, строкой, возвращенной делегатом `MatchEvaluator`.

`Split` – разделяет входную строку в массив подстрок в позициях, определенных шаблоном регулярного выражения

Вернемся к нашему примеру с регулярным выражением и рассмотрим работу этих методов на его примере. С методом `IsMatch` мы уже познакомились.

Поиск первого вхождения регулярного выражения в строку (метод `Match`)

```
string pattern = @"\s*(\w*н{2}\w*)\s*";
string input = "Анна наполнила ванну";
Regex regex = new Regex(pattern);
Match match = regex.Match(input);
if (match.Success) //нашли совпадение
{
Console.WriteLine($"Первое совпадение {match.Value}");
match = match.NextMatch();
while (match.Success)
{
Console.WriteLine($"Следующее совпадение {match.Value}");
match = match.NextMatch();
}
}
```

Итак, создали объект типа `Regex`, передав в конструкторе регулярное выражение. Затем нашли первое совпадение, используя метод `Match`. Этот метод вернул нам одноименный класс

Match, содержащий информацию о найденном совпадении. Далее, воспользовались уже методом класса Match.NextMatch и прошлись дальше по тексту, получив все совпадения с регулярным выражением. Но устраивать цикл не обязательно, так как все совпадения можно получить с помощью другого метода Regex.

Получение всех вхождений регулярного выражения в текст (метод Matches)

```
string pattern = @"\s*(\w*н{2}\w*)\s*";
string input = "Анна наполнила ванну";
Regex regex = new Regex(pattern);
MatchCollection collection = regex.Matches(input);
foreach (Match match in collection)
{
Console.WriteLine($"Совпадение {match.Value}");
GroupCollection groupCollection = match.Groups;
foreach (Group group in groupCollection)
{
Console.WriteLine($"\tЗначения группы: Индекс {group.Index}
\tИмя {group.Name} \tЗначение {group.Value} \tРазмер
{group.Length}");
}
}
```

В этом примере получаем сразу все вхождения регулярного выражения в текст, используя метод Matches. Затем проходимся по всей коллекции совпадений и считываем информацию о группах (GroupCollection) в каждом совпадении. Обратите внимание, что в регулярном выражении у нас обозначена одна группа, в то время как Regex будет создавать две группы (рис. 19.4):

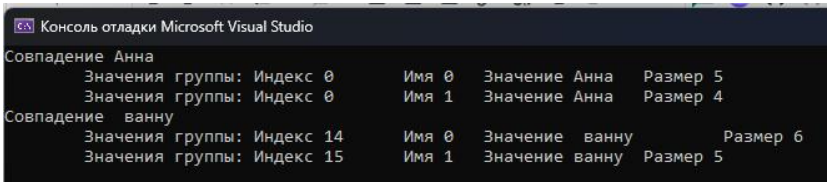


Рисунок 19.4. Результаты с найденными совпадениями

По полученным данным нетрудно понять, какие свойства за что отвечают в группе:

Group.Index – индекс символа в строке с которого начинается значение группы.

Group.Name – порядковый номер группы в коллекции.

`Group.Value` – значение группы.

`Group.Length` – длина значение в символах.

В первую группу (с индексом `0`) попадает строка, содержащая всё совпадение, то есть, в нашем случае строка вместе со всеми пробелами.

Замена подстрок в строке с использованием `Regex` (метод `Replace`).

Первый способ – замена строки на другую строку:

```
string pattern = @"\s*(\w*н{2}\w*)\s*";
string input = "Анна наполнила ванну";
Regex regex = new Regex(pattern);
string result = regex.Replace(input, "замена");
Console.WriteLine(result);//заменанаполнилазамена
```

Как видим, заменились все совпадения (не значения групп), и строка превратилась в один сплошной набор символов. Второй способ замены более гибкий и позволяет перед заменой произвести анализ того, что планируется заменить:

```
static string GetText(Match m)
{
string x = m.ToString();//получаем значение группы
if (char.IsUpper(x.Trim()[0])) //слово начинается с большой
буквы - это имя
{
return "Маша ";
}
else
return " ведро";
}
static void Main(string[] args)
{
string pattern = @"\s*(\w*н{2}\w*)\s*";
string input = "Анна наполнила ванну";
Regex regex = new Regex(pattern);
string result = regex.Replace(input, new
MatchEvaluator(GetText));
Console.WriteLine(result);
}
}
```

Тут мы воспользовались одной из перегруженных версий метода `Replace` и передали во втором параметре метод, который анализирует очередное совпадение с регулярным выражением и предлагает результат замены. Так, если в совпадении первый символ представлен заглавной буквой, то считаем, что перед нами

имя и его меняем на другое имя, иначе – меняем слово «ванна» на «ведро». В результате получим строку: «Маша наполнила ведро».

Разделение строки на подстроки по регулярному выражению (метод Split):

```
string pattern = @"\s*(\w*н{2}\w*)\s*";
string input = "Анна наполнила ванну";
Regex regex = new Regex(pattern);
string[] strings = regex.Split(input);
foreach (string s in strings)
{
Console.WriteLine(s);
}
```

В результате получим следующий массив строк:

```
Анна
наполнила
ванну
```

Причем массив будет состоять не из трех, а из пяти элементов, где первый и последний элементы – пустые строки.

Задания и порядок выполнения работы

Задание 1. Изучить возможности ресурса https://regex101.com/ по работе с регулярными выражениями в качестве регулярных выражений использовать примеры из теоретического материала.

Задание 2. Написать программу для замены всех URL в текстовом файле на ссылку “[URL]”.

Задание 3. Написать программу для подсчета количества слов, начинающихся с заглавной буквы, в введенной строке.

Задание 4. Написать программу для извлечения всех чисел из введенной строки и суммирования их.

Задание 5. Написать программу для проверки, является ли введенный пользователем пароль допустимым, согласно следующим правилам:

– должен содержать хотя бы одну цифру;

– должен содержать хотя бы один символ верхнего регистра;

– должен содержать хотя бы один специальный символ (например, @, #, $).

Задание 6. Написать программу для проверки корректности ввода серии и номера паспорта РФ.

Контрольные вопросы

1. Что такое регулярное выражение и для чего оно используется в программировании?
2. Какие метасимволы используются в регулярных выражениях C#?
3. Как создать регулярное выражение для проверки наличия email-адреса в строке?
4. Опишите синтаксис регулярных выражений в C#.
5. Каким образом можно использовать регулярные выражения в сочетании с методом String.Split()?
6. Напишите регулярное выражение для поиска всех URL в тексте.
7. Как найти все числа в строке с помощью регулярного выражения в C#?
8. Как проверить, начинается ли строка с заглавной буквы, с помощью регулярного выражения?
9. Как использовать группы в регулярных выражениях для извлечения данных из строки?
10. Опишите различия между классами Regex и Match в C# для работы с регулярными выражениями.

Лабораторная работа № 20

Тема: **«Разработка приложений с графическим интерфейсом. Изменение свойств компонентов в процессе выполнения приложения»**.

Цель: изучение принципов работы оператора цикла с параметром и циклов с условием в языке C#, а также отработка навыков их применения для написания программ.

Краткие теоретические сведения

Элемент управления Windows Forms Button позволяет пользователю щелкнуть его для выполнения действия. При щелчке кнопки мышью элемент управления выглядит так, как будто его нажимают и отпускают. Всякий раз, когда пользователь нажимает кнопку, вызывается обработчик событий Click. Код помещается в обработчик событий Click для выполнения любого необходимого действия.

Текст, отображаемый на кнопке, содержится в свойстве Text. Если текст превышает ширину кнопки, он перенесется на следующую строку. Однако он будет обрезан, если элемент управления не может вместить его общую высоту.

1. Свойства элементов управления

В элементах управления Windows Forms обычно отображается текст, связанный с их основной функцией. Например, в элементе управления Button обычно отображается заголовок, указывающий, какое действие выполняется при нажатии кнопки. С помощью свойства Text можно задавать или получать текст для всех элементов управления. Шрифт можно менять с помощью свойства Font.

```
button1.Text = "Click here to save changes";
button1.Font = new Font("Arial", 10, FontStyle.Bold,
GraphicsUnit.Point);
```

Все элементы управления наследуются от класса Control и поэтому имеют ряд общих свойств:

Anchor – определяет, как элемент будет растягиваться;

BackColor – определяет фоновый цвет элемента;

`BackgroundImage` – определяет фоновое изображение элемента;

`ContextMenu` – контекстное меню, которое открывается при нажатии на элемент правой кнопкой мыши;

`Cursor` – представляет, как будет отображаться курсор мыши при наведении на элемент;

`Dock` – задает расположение элемента на форме;

`Enabled` – определяет, будет ли доступен элемент для использования. Если это свойство имеет значение `False`, то элемент блокируется;

`Font` – устанавливает шрифт текста для элемента;

`ForeColor` – определяет цвет шрифта;

`Location` – определяет координаты верхнего левого угла элемента управления;

`Name` – имя элемента управления;

`Size` – определяет размер элемента;

`Width` – ширина элемента;

`Height` – высота элемента;

`TabIndex` – определяет порядок обхода элемента по нажатию на клавишу `Tab`;

`Tag` – позволяет сохранять значение, ассоциированное с этим элементом управления.

2. Задание клавиши доступа для элемента управления

Свойство `Text` может содержать клавишу доступа, которая позволяет пользователю «щелкнуть» элемент управления, нажав клавишу `ALT` вместе с клавишей доступа.

Клавиша доступа представляет собой подчеркнутый символ в тексте меню, пункте меню или метке элемента управления, например кнопки. С помощью клавиши доступа пользователь может «нажать» кнопку, нажав клавишу `ALT` в сочетании с предопределенной клавишей доступа. Например, если кнопка запускает процедуру печати формы и поэтому ее свойство `Text` имеет значение «`Print`», добавление амперсанда перед буквой «`P`» приводит к тому, что буква «`P`» будет подчеркнута в тексте

кнопки во время выполнения. Пользователь может выполнить команду, связанную с кнопкой, нажав сочетание клавиш ALT+P.

```
button1.Text = "&Print";
```

3. Элемент управления ListBox

Элемент управления ListBox в Windows Forms отображает список элементов, из которого пользователь может выбирать один элемент или несколько. Если общее число элементов превышает число, которое можно отобразить, полоса прокрутки автоматически добавляется в элемент управления ListBox. Если для свойства MultiColumn задано значение true, элементы в списке отображаются в нескольких столбцах с горизонтальной полосой прокрутки. Если для свойства MultiColumn задано значение false, элементы в списке отображаются в одном столбце с вертикальной полосой прокрутки. Если для ScrollAlwaysVisible задано значение true, полоса прокрутки появляется независимо от количества элементов. Свойство SelectionMode определяет, сколько элементов списка можно выбрать за раз.

Свойство SelectedIndex возвращает целочисленное значение, соответствующее первому выбранному элементу в списке. Вы можете программно изменить выбранный элемент, изменив значение SelectedIndex в коде. Соответствующий элемент в списке будет выделен в форме Windows Forms. Если элемент не выбран, значение SelectedIndex равно -1. Если выбран первый элемент в списке, значение SelectedIndex равно 0. При выборе нескольких элементов значение SelectedIndex отражает выбранный элемент, который отображается в списке. Свойство SelectedItem похоже на SelectedIndex, но возвращает сам элемент, обычно строковое значение. Свойство Count отражает количество элементов в списке, а значение свойства Count всегда на один больше, чем наибольшее возможное значение SelectedIndex, так как SelectedIndex отсчитывается от нуля.

Чтобы добавить или удалить элементы в элементе управления ListBox, используйте метод Add, Insert, Clear или

Remove. Кроме того, можно добавить элементы в список с помощью свойства Items во время разработки. Например

```
listBox1.Items.Add("Привет");
```

Чтобы удалить все элементы из коллекции, вызовите метод Clear:

```
listBox1.Items.Clear();
```

Чтобы вставить строку или объект в нужную позицию в списке, используйте метод Insert:

```
listBox1.Items.Insert(0, "Copenhagen");
```

Задания и порядок выполнения работы

Задание 1. Создайте форму, которая содержит три кнопки – Print, Open, File, которые можно вызвать, используя соответствующие комбинации клавиш ALT+P, ALT+O, ALT+F.

Задание 2. Напишите программу, отображающую окно, в котором внутренняя часть закрашена желтым цветом. При наведении курсора на область окна цвет фона меняется на зеленый. При щелчке мышью размеры окна должны увеличиваться на 10%.

Задание 3. Напишите программу, в которой отображается окно с текстовой меткой и тремя кнопками. В текстовой метке содержится число (начальное значение – нулевое). Щелчок по одной из меток приводит к увеличению значения числа на единицу. Щелчок по другой кнопке приводит к уменьшению значения числа на единицу. Щелчок по третьей кнопке приводит к закрытию окна.

Задание 4. Напишите программу, в которой открывается окно с раскрывающимся списком. Список содержит названия цветов (красный, желтый, зеленый и так далее). Также окно содержит область, закрашенную тем цветом, который выбран в списке. При выборе в списке нового цвета область закрашивается этим цветом автоматически.

Контрольные вопросы

1. Какое свойство элемента управления Button определяет его размер и положение на форме?

2. Каким образом можно изменить текст на кнопке во время выполнения программы?

3. Каким свойством обладает ListBox, которое позволяет отображать только определенное количество элементов?

4. Каким образом можно добавить новые элементы в ListBox во время выполнения программы?

5. Каким свойством Label можно управлять его размерами и положением на форме?

6. Каким образом можно изменять содержимое Label во время выполнения программы?

7. Можно ли изменять цвет текста на кнопке? Если да, то каким свойством это можно сделать?

8. Можно ли добавить границу вокруг ListBox? Если да, то какое свойство нужно использовать?

9. Можно ли изменить шрифт текста на метке? Если да, то с помощью какого свойства это можно сделать?

10. Можно ли сделать кнопку невидимой? Если да, то как это можно осуществить?

Лабораторная работа № 21

Тема: **«Работа с полями для ввода, списками, выпадающими списками»**.

Цель: изучить особенности работы с различными элементами управления, такими как поля для ввода текста, списки и выпадающие списки, в приложениях с графическим интерфейсом, научиться создавать эффективные и удобные интерфейсы с использованием данных элементов.

Краткие теоретические сведения

1. Элемент управления TextBox

Текстовые поля Windows Forms используются для получения входных данных от пользователя или для отображения текста. Элемент управления TextBox обычно используется для редактируемого текста, хотя для него также можно установить разрешения только для чтения. Текстовые поля могут отображать несколько строк, регулировать размер текста по размеру элемента управления и добавлять основное форматирование. Элемент управления TextBox предоставляет один стиль формата для текста, отображаемого или введенного в элемент управления.

Текст, отображаемый элементом управления, содержится в свойстве Text. По умолчанию в текстовом поле можно ввести до 2048 символов. Если для свойства Multiline задано значение true, можно ввести до 32 КБ текста. Свойство Text можно задать во время разработки с помощью окна свойств, во время выполнения в коде или путем ввода данных пользователем во время выполнения. Текущее содержимое текстового поля можно получить во время выполнения, прочитав свойство Text. Например:

```
textBox1.Text = "Это элемент управления TextBox";
```

Задайте для свойства ReadOnly элемента управления TextBox значение true. Если для свойства задано значение true, пользователи по-прежнему могут прокручивать и выделять текст в текстовом поле, но не изменять его. Команда Копировать работает в текстовом поле, но команды Вырезать и Вставить – нет.

Использование свойства PasswordChar в текстовом поле поможет убедиться, что другие пользователи не смогут увидеть пароль пользователя, когда пользователь вводит его.

```
// Set to no text.
textBox1.Text = "";
// The password character is an asterisk.
textBox1.PasswordChar = '*';
// The control will allow no more than 14 characters.
textBox1.MaxLength = 14;
```

2. Элемент управления RichTextBox

Элемент управления RichTextBox в Windows Forms используется для отображения, ввода и обработки текста с форматированием. Функции элемента управления RichTextBox аналогичны функциям элемента управления TextBox, однако он также позволяет отображать шрифты, цвета и ссылки, загружать текст и встроенные изображения из файла и находить указанные символы. Элемент управления RichTextBox обычно используется для работы с текстом и отображения функций, аналогичных текстовым редакторам, таким как Microsoft Word. Как и элемент управления TextBox, RichTextBox может отображать полосы прокрутки; но в отличие от элемента управления TextBox его значение по умолчанию подразумевает отображение горизонтальных и вертикальных полос прокрутки при необходимости, кроме того, он имеет дополнительные параметры полосы прокрутки.

3. Элемент управления ComboBox

Элемент управления Windows Forms ComboBox служит для отображения данных в поле с раскрывающимся списком. По умолчанию элемент управления ComboBox состоит из двух частей. Верхняя часть – это текстовое поле, в котором пользователь может ввести элемент списка. Вторая часть – это список элементов, из которых пользователь может выбрать один.

Элементы управления ComboBox и ListBox имеют аналогичное поведение, а в некоторых случаях могут быть

взаимозаменяемыми. Однако порой для выполнения задачи наилучшим образом подходит только один из них.

Как правило, поле со списком используется, если имеется список предлагаемых вариантов, а список – если требуется ограничить входные данные содержимым списка.

Если элемент управления не привязан к данным, его можно отсортировать. Для этого задайте свойству `Sorted` значение `true`.

Для получения значения списка по индексу `i` можно использовать код

```
comboBox1.Items[i].ToString()
```

Задания и порядок выполнения работы

Задание 1. Напишите программу, в которой открывается окно с полем ввода. При вводе текста в окно этот текст автоматически дублируется в текстовой метке. В окне должны быть две опции, которые позволяют применять к тексту в метке жирный и курсивный стили.

Задание 2. Напишите программу, в которой отображается окно с двумя текстовыми полями. Предполагается, что в эти текстовые поля вводятся целочисленные значения. Кроме полей, в окне размещена метка, в которой содержится информация о том, какое из двух чисел больше/меньше или что числа равны друг другу. Информация в метке обновляется автоматически при изменении содержимого полей. Если хотя бы в одном из полей указано не число, метка должна содержать информацию об этом.

Задание 3. Напишите программу, в которой отображается окно со списком выбора. В списке выбора представлены названия шрифтов. Также окно содержит раскрывающийся список с названиями цветов (красный, зеленый, синий и так далее). Окно содержит область с текстом. При выборе цвета или названия шрифта этот цвет или шрифт применяются для отображения текста.

Задание 4. Напишите программу, в которой отображается окно с закрашенной областью. Для этой области есть контекстное меню с названиями цветов (красный, желтый, зеленый и так

далее). При выборе команды из контекстного меню область закрашивается соответствующим цветом.

Задание 5. Напишите программу, в которой отображается окно, представляющее собой, арифметический калькулятор.

Задание 6. Напишите программу, в которой отображается окно с главным меню и областью с текстом. Текст содержит информацию о названии, стиле и размере шрифта, которым отображается текст. В меню есть пункты для выбора названия шрифта, стиля шрифта и размера шрифта. При выборе команды из меню соответствующая характеристика применяется для отображения текста, а также с учетом новых параметров шрифта меняется сам текст.

Контрольные вопросы

1. Как добавить элемент в ListBox в C#?
2. Чем отличается ListBox от ComboBox?
3. Можно ли установить цвет фона ListBox в C#? Если да, то как?
4. Можно ли изменить размер шрифта текста в ListBox?
5. Можно ли в ComboBox выбрать несколько значений? Если да, то как это сделать?
6. Можно ли поменять цвет текста в ComboBox? Если да, то как?
7. Можно ли менять свойства шрифта в ListBox и ComboBox? Если да, то какие свойства можно менять и как?
8. Можно ли использовать разные цвета фона для разных элементов в ListBox или ComboBox?
9. Какие свойства можно настроить для ListBox и ComboBox через дизайнер форм в Visual Studio?
10. Можно ли скрыть или отобразить определенные элементы в ListBox или ComboBox во время выполнения программы?

Лабораторная работа № 22

Тема: **«Основы работы с делегатами в C#»**

Цель: изучить основы работы с делегатами в C#, способы применения и преимущества использования делегатов при разработке программ на C#.

Краткие теоретические сведения

Делегат в C# – это ссылочный тип данных, который предоставляет ссылки на методы. Так, если переменная в C# описывает данные и их тип, то делегат описывает метод, его параметры и тип возвращаемых данных.

1. Использование делегатов

Для объявления делегата в C# используется ключевое слово delegate. Например,

```
delegate void EventHandler();
```

Нами объявлен делегат, который описывает метод без каких-либо параметров и который ничего не возвращает. Можем объявить любой делегат, например, такой:

```
delegate double Sum(double x, double y);
```

После того, как объявили делегат, он может указывать на любой метод, имеющий точно такую же сигнатуру, как и у делегата.

Рассмотрим работу с созданными выше делегатами на примере:

```
using System;
namespace Delegates
{
    class Program
    {
        delegate void EventHandler();
        delegate double Sum(double x, double y);
        //метод на который будет ссылаться делегат EventHandler
        static void SayHello()
        {
            Console.WriteLine("Hello World!");
        }
        //метод на который будет ссылаться делегат Sum
        static double Calculate(double x, double y)
        {
            return x + y;
```

```
        }
        //использование делегатов
        static void Main(string[] args)
        {
            EventHandler handler = SayHello;//создаем
переменную делегата и присваиваем ей значение
            handler(); //вызываем метод
            Sum sum = Calculate;//создаем переменную делегата и
присваиваем ей значение
            double result = sum(15.4, 0.6);//вызываем метод
            Console.WriteLine($"Сумма двух чисел равна
{result}");
        }
    }
}
```

Тут объявили два делегата (EventHandler и Sum), затем объявили две переменные с типами наших делегатов (handler и sum) и присвоили им значение – указали на методы SayHello и Calculate, которые должны вызываться. После этого вызвали методы и получили результаты выполнения методов в консоль.

И стоит отметить важный момент: если можно работать с делегатами также, как и с любыми типами данных в C# – объявлять переменные типа делегата и присваивать им значение, то можно также передавать такие переменные типа делегата в качестве параметров других методов. Например:

```
using System;
namespace Delegates
{
    class Program
    {
        delegate void EventHandler();

        //метод на который будет ссылаться делегат EventHandler
        static void SayHello()
        {
            Console.WriteLine("Hello World!");
        }
        //метод на который будет ссылаться делегат EventHandler
        static void SayGoodbye()
        {
            Console.WriteLine("Goodbye World!");
        }
        //метод, принимающий делегат в качестве параметра
        static void PrintString(EventHandler handler)
```

```
        {
            handler();
        }
        //использование делегатов
        static void Main(string[] args)
        {
            EventHandler handler;
            if (DateTime.Now.Hour < 12)
                handler = SayHello;//присваиваем значение
            else
                handler = SayGoodbye;//присваиваем значение
            PrintString(handler);

        }
    }
}
```

Метод `PrintString` принимает в качестве параметра переменную типа делегата `EventHandler`.

Создавать объекты делегата можно не только прямым присвоением переменной метода, как это делали выше, но и с использованием конструктора:

```
handler = new EventHandler(SayHello);
```

Этот способ полностью идентичен тому, который использовался в примерах.

2. Добавление методов в делегат

С переменными типа делегата можно выполнять операции сложения и вычитания. Добавление методов в делегат – это ещё одна важная особенность этого типа данных в C#. При добавлении метода в делегат он попадает в специальный список `InvocationList` и при вызове делегата каждый метод из этого списка последовательно выполняется. Например:

```
using System;
namespace Delegates
{
    class Program
    {
        delegate void EventHandler();

        //метод на который будет ссылаться делегат EventHandler
        static void SayHello()
        {
```

```
            Console.WriteLine("Hello World!");
        }
        //метод на который будет ссылаться делегат EventHandler
        static void SayGoodbye()
        {
            Console.WriteLine("Goodbye World!");
        }
        //использование делегатов
        static void Main(string[] args)
        {
            EventHandler handler;
            Console.WriteLine("Добавляем метод в делегат");
            handler = new EventHandler(SayHello);//присваиваем
значение
            handler += SayGoodbye;//прибавили метод
            handler();
            Console.WriteLine("Исключаем метод из делегата");
            handler -= SayHello;
            handler();

            handler -= SayGoodbye;
            Console.WriteLine("После    это    строки    будет
исключение так как переменная делегата перестанет ссылаться на
методы");
            handler();
        }
    }
}
```

В этом примере переменная `handler` перед первым вызовом `handler()` ссылается сразу на два метода, затем исключаем один из методов (отнимаем) из списка и выводим в консоль уже одну строку и, наконец, на третьем шаге удаляем все ссылка на методы и получаем исключение, так как переменная более не указывает ни на один метод.

Более того, можем применять операции сложения непосредственно к переменным делегатов, например:

```
static void Main(string[] args)
{
    EventHandler HelloHandler = new EventHandler(SayHello);
    EventHandler GoodbyeHandler = SayGoodbye;
    EventHandler eventHandler = HelloHandler + GoodbyeHandler;
    eventHandler();
}
```

Так как типы делегата являются производными от `System.Delegate`, то в делегате можно вызывать методы и свойства, определенные этим классом. До сих пор делегат вызывался как обычный метод, используя же возможности `System.Delegate` можно воспользоваться для вызова специальным методом `Invoke()`. Ниже представлены два равнозначных способа вызова делегата:

```
EventHandler HelloHandler = new EventHandler(SayHello);
HelloHandler();
HelloHandler.Invoke();
```

Задания и порядок выполнения работы

Задание 1. Выполнить все примеры из лекции.

Задание 2. Напишите программу, в которой объявляется делегат для методов с двумя аргументами (символ и текст) и целочисленным результатом. В главном классе необходимо описать два статических метода. Один статический метод результатом возвращает количество вхождений символа (первый аргумент) в текстовую строку (второй аргумент). Другой метод результатом возвращает индекс первого вхождения символа (первый аргумент) в текстовую строку (второй аргумент) или значение `-1`, если символ в текстовой строке не встречается. В главном методе создать экземпляр делегата и с помощью этого экземпляра вызвать каждый из статических методов.

Задание 3. Напишите программу, в которой объявляется делегат для методов с символьным аргументом, не возвращающих результат. Опишите класс, в котором должно быть символьное поле и метод, позволяющий присвоить значение символьному полю объекта. У метода один символьный аргумент, и метод не возвращает результат. Создайте массив объектов данного класса. Создайте экземпляр делегата. В список вызовов этого делегата необходимо добавить ссылки на метод (присваивающий значение символьному полю) каждого объекта из массива. Проверьте результат вызова такого экземпляра делегата.

Задание 4. Напишите программу, в которой объявляется делегат для работы с методами, имеющими целочисленный аргумент и целочисленный результат. Опишите класс с

индексатором (доступен только для считывания значения). Индексатор результатом должен возвращать ссылку на экземпляр делегата. Экземпляр делегата ссылается на метод, у которого целочисленный аргумент. Результатом метод возвращает целочисленное значение, получающееся возведением аргумента метода в степень, определяемую индексом объекта. Общий эффект такой: если некоторый объект `obj` класса проиндексировать с неотрицательным индексом `k` и в круглых скобках указать аргумент `n` (команда вида `obj[k](n)`), то результатом такого выражения должно быть значение `n` в степени `k`.

Контрольные вопросы

1. Что такое делегаты в C# и для чего они используются?
2. Чем отличается делегат от события?
3. Опишите синтаксис объявления делегата.
4. Каким образом происходит вызов делегата?
5. Может ли делегат вызывать несколько методов одновременно?
6. Что такое многоадресный делегат?
7. Опишите принцип работы события в C#.
8. Для чего используются анонимные методы и лямбда-выражения при обработке событий?
9. Можно ли отменить регистрацию события в C#? Если да, то как это сделать?
10. Приведите примеры использования делегатов и событий в реальных проектах на C#.

Лабораторная работа № 23

Тема: **«Создание пользовательских классов с использованием делегатов и событий»**

Цель: научиться создавать пользовательские классы в C# с использованием делегатов и событий, изучить особенности применения делегатов и событий при наследовании и абстрагировании данных, а также рассмотреть практические примеры использования разработанных классов в различных сценариях программирования.

Краткие теоретические сведения

Одно из преимуществ делегата состоит в том, что, используя его можно делегировать выполнение кода абсолютно любому методу с той же сигнатурой, что и у делегата.

1. Использование делегатов в C# на примере своего класса

Известно, что такое класс и как его создавать, поэтому наглядное применение делегатов в C# наиболее выгодно продемонстрировать именно при использовании их в классах. Допустим, пишем приложение для работы со списком сотрудников организации. Для этого у нас в программе определен класс `Person`

```
public class Person
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public string Department { get; private set; }
    public byte Age { get; set; }
    public void ChangeDepartmnet(string newDepartment)
    {
        Department = newDepartment;
    }
}
```

У класса сотрудника определены свойства: имя, фамилия и отдел, в котором он работает. При этом, свойство `Department` у нас доступно извне только для чтения и чтобы изменить название отдела в котором работает сотрудник необходимо вызвать метод `ChangeDepartmnet()`. Теперь попробуем ответить на такой вопрос: что нам (как пользователям этого класса) потребуется сделать,

если мы захотим изменить логику метода ChangeDepartmnet() , например, вывести в консоль строку с информацией о том, что у сотрудника изменился отдел? Правильно – перепишем тело нашего метода. А что, если этот класс будет находится в какой-нибудь библиотеке классов, и мы не имеем доступа к исходному коду этого метода? И вот здесь-то и раскрывается преимущество использования делегата в C# –можно делегировать выполнение необходимых нам действий другому методу. Перепишем наш класс следующим образом:

```
public class Person
{
    public delegate void PrintString(string str); //объявили
делегат
    PrintString print; //объявили переменную делегата

    public string Name { get; set; }
    public string Surname { get; set; }
    public string Department { get; private set; }
    public byte Age { get; set; }
    public void ChangeDepartmnet(string newDepartment)
    {
        Department = newDepartment;
        print?.Invoke($"Отдел работы сотрудника {Name}
{Surname} изменен на {newDepartment}");
    }
    public void Register(PrintString printString)
    {
        print += printString;
    }
    public void Unregister(PrintString printString)
    {
        print -= printString;
    }
}
```

Во-первых, объявили делегат PrintString:

```
  public delegate void PrintString(string str); //объявили
делегат
```

Во-вторых, объявили переменную типа делегата:

```
PrintString print; //объявили переменную делегата
```

В-третьих, так как давать прямой доступ к переменным класса извне не рекомендуется, то были созданы два метода – для

добавления метода к делегату (Register) и снятия метода (Unregister).

В-четвертых, переписали метод ChangeDepartmnet в котором производим вызов делегата в том случае, если он не равен null, то есть переменная делегата ссылается хотя бы на один метод.

Теперь протестируем наш класс:

```
class Program
{
    //использование делегатов
    static void Main(string[] args)
    {
        Person person = new Person
        {
            Name = "Вася",
            Surname = "Пупкин"
        };
        person.ChangeDepartmnet("Курьерский");
        person.Register(Show);
        person.ChangeDepartmnet("Общий");
    }
    public static void Show(string message)
    {
        Console.WriteLine(message);
    }
}
```

Добавили метод Show в делегат после первого вызова метода ChangeDepartmnet. Следовательно, в консоли увидим только такую строку:

```
Отдел работы сотрудника Вася Пупкин изменен на Общий
Последовательный вызов методов делегата
```

Опять же, никто нам не запрещает регистрировать несколько методов для делегата, объявленного в нашем классе Person и эти методы последовательно будут вызываться. Перепишем код нашей программы следующим образом:

```
class Program
 {
     //использование делегатов
     static void Main(string[] args)
     {
         Person person = new Person
         {
             Name = "Вася",
```

```
            Surname = "Пупкин"
        };
        person.Register(Show);
        person.Register(ShowRed);
        person.ChangeDepartmnet("Общий");
    }
    public static void Show(string message)
    {
        Console.ResetColor();
        Console.WriteLine(message);
    }
    public static void ShowRed(string message)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
    }
}
```

Теперь переменная делегата содержит ссылки на два метода – Show и ShowRed. Таким образом, в консоль выведется две строки – одна белого цвета, а вторая – красного. При этом, обратите, внимание, что исходный код класса Person не был изменен.

События в C# позволяют классу или объекту уведомлять другие классы или объекты о возникновении каких-либо ситуаций. События активно используются в Windows-приложениях. Класс, который порождает (отправляет) событие, называется издателем, а классы, обрабатывающие (принимающие) событие, называются подписчиками. Соответственно, на одно и то же событие могут подписываться несколько подписчиков.

2. События в C#

Возвращаясь к классу Person, который использовали, когда разбирали тему использования делегатов. Итоговый класс имел следующий вид:

```
public class Person
{
    public delegate void PrintString(string str); //объявили
делегат
    PrintString print; //объявили переменную делегата

    public string Name { get; set; }
    public string Surname { get; set; }
    public string Department { get; private set; }
```

```
    public byte Age { get; set; }
    public void ChangeDepartmnet(string newDepartment)
    {
        Department = newDepartment;
        print?.Invoke($"Отдел работы сотрудника {Name}
{Surname} изменен на {newDepartment}");
    }
    public void Register(PrintString printString)
    {
        print += printString;
    }
    public void Unregister(PrintString printString)
    {
        print -= printString;
    }
}
```

Использование метода `ChangeDepartmnet` в данном случае хоть и возможно, но избыточно. Более логично, в данном случае, позволить пользователю класса менять свойство `Department`, как и другие свойства и, при этом, в своей программе «отлавливать» момент изменения этого свойства и выводить результат в консоль. И здесь-то нам и помогут события в C#.

В C# событие определяется следующим образом:

```
event delegateType EventName;
```

где

`event` – ключевое слово, которое сообщает нам, что перед нами событие

`delegateType` – это тип делегата. Делегат описывает то, как должен выглядеть метод в подписчике, который будет обрабатывать событие, а также то, какие параметры необходимо передавать подписчику. То есть, делегат, образно выражаясь, представляет из себя некий договор между издателем и подписчиком как они будут между собой общаться.

`EventName` – название события

В библиотеке классов .NET события основываются на делегате `EventHandler` и базовом классе `EventArgs`. Делегат `EventHandler` выглядит следующим образом:

```
public delegate void EventHandler(object? sender, EventArgs e);
```

`sender` – определяет класс или объект, который породил событие (издатель)

e – класс, содержащий параметры, передаваемые подписчику.

Технически, зная то, что из себя представляет делегат в C# и как он определяется, можно определить делегат для нашего события в любом удобном для нас виде, хоть так:

```
public delegate void MyEventHandler(string str, DateTime
date, out int count);
```

и затем объявить вот такое событие:

```
public event MyEventHandler MySuperEvent;
```

Такой код будет работать, НО правильным и хорошим тоном считается придерживаться следующего правила: делегат события содержит два параметра – первый предоставляет подписчику информацию об издателе, а второй – наследник класса EventArgs передает подписчику необходимые параметры. Попробуем переписать наш класс Person и создать свое первое событие в C#.

3. Создание события в C

Перепишем наш класс Person следующим образом:

```
public class Person
{
    //событие при смене отдела
    public event EventHandler ChangeDepartment;
    public string Name { get; set; }
    public string Surname { get; set; }

    private string department;
    public string Department
    {
        get { return department; }
        set
        {
            if (department != value)
            {
                department = value;
                ChangeDepartment?.Invoke(this, new
EventArgs());
            }

        }
    }
    public byte Age { get; set; }
}
```

Обратим внимание на следующие моменты:

Мы объявили событие типа `EventHandler` с названием `ChangeDepartment` – оно будет порождаться, когда у объекта будет изменяться название отдела

Свойства `Department` вместо сокращенной формы записи теперь имеет блоки `get` и `set`. При этом, в блоке `set` проверяется действительно ли новое название отдела не совпадает с текущим и только, если новое название не совпадает со старым меняется значение у `department` и вызывается метод делегата.

Теперь можем подписаться на это событие и получать уведомления от издателя. Сделать это можно, например, так:

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person
        {
            Name = "Вася",
            Surname = "Пупкин",
            Department = "Курьерский"
        };
        person.ChangeDepartment += EventHandler;
        person.Department = "Общий"; //здесь событие сработает
        person.Department  =  "Общий";  //здесь  событие  не
сработает
    }
    public static void EventHandler(object sender, EventArgs e)
    {
        Console.WriteLine($"У работника сменился отдел на
{((Person)sender).Department}");
    }
}
```

То есть, создали объект класс `Person`, и назначили в качестве обработчика события `ChangeDepartment` метод `EventHandler`. Теперь, если запустить приложение, то можно увидеть, что событие сработает ровно один раз:

```
У работника сменился отдел на Общий
```

4. Подписка на события в C#

На любое событие в C# может быть подписано любое количество подписчиков. Более того, один и тот же класс может подписаться любое количество раз на событие:

```
class Program
 {
     static void Main(string[] args)
     {
         Person person = new Person
         {
             Name = "Вася",
             Surname = "Пупкин",
             Department = "Курьерский"
         };
         person.ChangeDepartment += EventHandler;
         person.ChangeDepartment += EventHandler_2;
         person.Department = "Общий"; //здесь событие сработает
     }
     public static void EventHandler(object sender, EventArgs e)
     {
         Console.WriteLine($"У работника сменился отдел на
{((Person)sender).Department}");
     }
     public static void EventHandler_2(object sender, EventArgs e)
     {
         Console.ForegroundColor = ConsoleColor.Red;
         Console.WriteLine($"У работника сменился отдел на
{((Person)sender).Department}");
         Console.ResetColor();
     }
 }
```

Как и в случае работы с обычными делегатами, методы EventHandler и EventHandler_2 будут вызваны последовательно:

```
У работника сменился отдел на Общий
У работника сменился отдел на Общий
```

5. Создание собственных делегатов событий в C#

В приведенном выше примере использовался уже имеющийся в .NET делегат события EventHandler, который возвращал нам, опять же, имеющийся в .NET C# класс EventArgs. По сути EventArgs – это пустой набор параметров и выглядит он следующим образом:

```
public class EventArgs
```

```
{
    public static readonly EventArgs Empty;
    public EventArgs()
    {
    }
}
```

Но что, если нам потребуется, чтобы событие `ChangeDepartment` сообщало нам не только факт смены отдела, но и то, какой отдел был ранее? Поможет создание собственного делегата события. Делается это следующим образом:

Во-первых, создаем класс-наследник `EventArgs` который будет содержать необходимые параметры. Например, можем определить такой класс:

```
public class ChangeDepartmentArgs : EventArgs
{
    public string OldDepartment { get; set; }
    public string NewDepartment { get; set; }
    public ChangeDepartmentArgs(string oldDepartment, string
newDepartment)
    {
        OldDepartment = oldDepartment;
        NewDepartment = newDepartment;
    }
}
```

Стоит обратить внимание на конструктор – в нем как раз передаем старое и новое название и присваиваем эти значения свойствам класса `OldDepartment` и `NewDepartment`.

Во-вторых, объявляем делегат для события:

```
public delegate void ChangeDepartmentHandler(object sender,
ChangeDepartmentArgs e);
```

В-третьих, определяем событие в классе:

```
public event ChangeDepartmentHandler ChangeDepartment;
```

В-четвертых, обеспечить внутри класса вызов события:

```
public class Person
{
    прочие свойства класса

    private string department;
    public string Department
    {
        get { return department; }
        set
        {
```

```
            if (department != value)
            {
                string old = department;
                department = value;
                ChangeDepartment?.Invoke(this,              new
ChangeDepartmentArgs(old, department));
            }
        }
    }
}
```

Остается только подписаться на событие и получать сообщения от издателя. Ниже показан код основной программы:

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person
        {
            Name = "Вася",
            Surname = "Пупкин",
            Department = "Курьерский"
        };
        person.ChangeDepartment += EventHandler;
        person.Department = "Общий"; //здесь событие сработает
    }
    public    static    void    EventHandler(object    sender,
ChangeDepartmentArgs e)
    {
        Console.WriteLine($"У  работника  сменился  отдел  с
{e.OldDepartment} на {e.NewDepartment}");
    }
}
```

Теперь класс `Person` содержит собственное событие, используя которое можно получать информацию об изменении свойства `Department`, включая и предыдущее название. В C# можно создавать события, которые могут отдавать (и получать) любое количество параметров.

Задания и порядок выполнения работы

Задание 1. Повторить все примеры, представленные в теоретическом материале.

Задание 2. Разработать классы `Student`, `Department`, `Kafedra`, которые иллюстрируют событийно-ориентированный подход.

Контрольные вопросы

1. Дайте определение понятию «делегат» в контексте языка программирования C#.

2. Объясните, как использовать делегаты для вызова методов с неизвестным сигнатурным типом.

3. Что такое событие в контексте объектно-ориентированного программирования и языка C#? Приведите примеры событий.

4. В чём разница между делегатом и событием в C#?

5. Как происходит подписка на события и отписка от них в C#?

6. Можно ли использовать делегат в качестве типа данных свойства класса в C#? Обоснуйте свой ответ.

7. Поясните, как работают анонимные методы в контексте обработки событий на языке C#.

8. Возможно ли наследование классов от делегатов в C#? Поясните свой ответ.

9. Опишите, как происходит вызов делегатов с точки зрения исполняющей системы C#.

10. Что такое `multicast`-делегаты и для чего они нужны? Приведите пример их использования.

Лабораторная работа № 24

Тема: **«Работа с перечислениями и структурами»**

Цель: получить знания о структурах данных и принципах работы с ними в языке программирования C#, изучив такие понятия, как перечисления и структуры, их особенности и сферы применения, а также научиться применять полученные знания на практике.

Краткие теоретические сведения

Перечисление – это тип данных, определяемый пользователем. Переменная, объявленная как относящаяся к типу перечисления, может принимать значение одной из целочисленных констант. Список этих констант фактически детерминирует (определяет) перечисление как тип.

1. Использование перечислений

Таким образом, определяя перечисление как тип данных, задается набор значений, которые может принимать переменная, относящаяся к данному типу. Значения, входящие в набор, целочисленные (по умолчанию типа `int`), но каждое имеет свое уникальное имя (то есть является константой).

Для использования перечисления в программе нужно сначала описать тип перечисления. После того как тип определен, его можно использовать (в качестве типа переменных, например). Ниже приведен шаблон для объявления перечисления:

```
enum имя {константа,константа,...,константа};
```

Начинается все с ключевого слова `enum`. Оно является индикатором того, что объявляется именно перечисление. После ключевого слова `enum` указывают имя перечисления, которое фактически представляет собой название создаваемого типа. Именно имя перечисления используется как идентификатор типа данных при объявлении переменных.

По умолчанию первая константа в списке получает значение `0`, вторая константа в списке получает значение `1`, и так далее: значение каждой следующей константы на единицу больше ее

предшественницы. Допустим, в программе при объявлении перечисления использована следующая команда:

```
enum Animals {Cat,Dog,Fox,Wolf,Bear};
```

Этой командой определяется перечисление `Animals`. Впоследствии идентификатор `Animals` можно использовать как обозначение для типа данных. Например, объявить переменную, относящуюся к типу перечисления `Animals`:

```
Animals animal;
```

Объявленная таким образом переменная `animal` относится к типу `Animals`, и это означает, что значением переменной может быть одна из констант, формирующих тип `Animals` (имеются в виду константы `Cat`, `Dog`, `Fox`, `Wolf` и `Bear`). Напомним, что константы, формирующие перечисление, являются целочисленными. Их значения – целые числа. В данном случае значение первой в списке `{Cat,Dog,Fox,Wolf,Bear}` константы `Cat` равно `0`, значение константы `Dog` равно `1`, и так далее – последняя (пятая по счету) в списке константа `Bear` имеет значение `4`. Константы из списка получают эти значения автоматически.

Также можно явно инициализировать константы, формирующие перечисления. В таком случае после имени константы указывается оператор присваивания `=` и значение константы. Причем не обязательно явно указывать значение для всех констант перечисления. Разрешается указать значения только некоторых констант. В таком случае для констант, значение которых явно не указано, действует прежнее правило: значение очередной константы на единицу больше значения предыдущей константы. Допустим, перечисление объявлено с помощью такой команды:

```
enum Animals: byte {Cat,Dog,Fox=100,Wolf,Bear=200};
```

В этом случае константа `Cat` реализуется с целочисленным значением `0`, константа `Dog` реализуется со значением `1`, константа `Fox` реализуется со значением `100`, значение константы `Wolf` равно `101`, а значение константы `Bear` равно `200`. Рассмотрим пример:

```
using System;
// Объявление перечисления:
enum Animals { Cat, Dog, Fox, Wolf, Bear };
```

```
// Объявление перечисления с инициализацией констант:
enum Coins {One=1,Two,Five=5,Ten=10,Fifty=50};
// Класс с главным методом:
class EnumDemo
{
    static void Main()
    {
        Console.WriteLine("В мире животных");
        // Переменная типа перечисления:
        Animals animal = Animals.Cat;
        Console.WriteLine("animal:  {0,-5}  или  {1}",  animal,
(int)animal);
        // Новое значение переменной:
        animal=Animals.Dog;
        Console.WriteLine("animal:  {0,-5}  или  {1}",  animal,
(int)animal);
        // Преобразование целого числа в значение
        // типа перечисления:
        animal=(Animals)2;
        Console.WriteLine("animal:  {0,-5}  или  {1}  ",  animal,
(int)animal);
        // Сумма переменной и целого числа:
        animal=animal+1;
        Console.WriteLine("animal:  {0,-5}  или  {1}",  animal,
(int)animal);
        // Применение операции инкремента:
        animal++;
        Console.WriteLine("animal:  {0,-5}  или  {1}",  animal,
(int)animal);
        Console.WriteLine("В мире финансов");
        // Переменная типа перечисления:
        Coins coin;
        // Объект с константами из перечисления:
        Array names=Enum.GetValues(typeof(Coins));
        // Перебор констант:
        for(int k=0;k<names.Length;k++){
        // Значение переменной:
        coin=(Coins)names.GetValue(k);
        Console.WriteLine("coin:   {0,-5}   или   {1}",   coin,
(int)coin);
    }
}}
```

В результате получим:

```
В мире животных
animal: Cat   или 0
animal: Dog   или 1
animal: Fox   или 2
```

```
animal: Wolf  или 3
animal: Bear  или 4
В мире финансов
coin: One   или 1
coin: Two   или 2
coin: Five  или 5
coin: Ten   или 10
coin: Fifty или 50
```

Числовые значения констант не всегда отличаются на единицу. В общем случае, удобно использовать специальные методы класса `Enum`. Сначала с помощью команды `Array names=Enum.GetValues(typeof(Coins))` создаем объект `names` класса `Array`, содержащий набор значений констант из перечисления `Coins`. Для этого из класса `Enum` вызывается статический метод `GetValues()`. Результатом метод возвращает ссылку на объект класса `Array`. Этот объект по свойствам напоминает массив и содержит, как отмечалось, значения, формирующие тип `Coins`. Аргументом методу `GetValues()` передается объект класса `Type`, который содержит информацию о перечислении `Coins`. Этот объект можно получить с помощью инструкции `typeof(Coins)`. Для перебора значений, формирующих тип перечисления `Coins`, запускается оператор цикла, в котором используем индексную переменную `k`. Количество элементов в объекте `names` вычисляется выражением `names.Length`, как для массива. Для считывания очередного значения, содержащегося в объекте `names`, используем метод `GetValue()`. Аргументом методу передается индекс считываемого элемента. Но поскольку значением выражения `names.GetValue(k)` является объектная ссылка класса `object`, то используем процедуру явного приведения типа. В итоге, получается команда `coin=(Coins)names.GetValue(k)`, которой значение присваивается переменной `coins`.

2. Структуры в C#

Структура `(struct)` в `C#` – это пользовательский тип данных, который используется наряду с классами и может содержать какие-либо данные и методы. Структурами также являются такие типы данных как `int`, `double` и т.д. Основное

отличие структуры (struct) от класса (class) заключается в том, что структура – это тип значений, а класс – это ссылочный тип.

Для того, чтобы объявить переменную типа структуры в C# используется ключевое слово struct:

```
struct имя_структуры
{
// элементы структуры
}
```

После ключевого слова struct следует имя структуры и далее, в фигурных скобках – элементы структуры (поля, методы и т.д.). Например, определим структуру, которая описывает точку в трехмерном пространстве:

```
public struct Point3D
{
public double X { get; set; }
public double Y { get; set; }
public double Z{ get; set; }
public override string ToString()
{
return $"({X},{Y},{Z})";
}}
```

Наша структура содержит три свойства – координаты X,Y,Z и один переопределенный метод ToString, который возвращает строку с координатами точки. Обращение к полям, свойствам и методам структуры, как и в случае с классами, происходит с использованием имени переменной, например:

```
Point.X = 100;
Point.Y = 100;
Point.Z = 100;
Console.WriteLine(Point.ToString());
```

где Point – это имя переменной типа Point3D.

Как и в случае с классами, структуры в C# можно создавать с использованием ключевого слова new:

```
Point3D Point = new Point3D();
```

После того, как структура создана, её полям и свойствам можно присваивать значения (см. в предыдущем пункте).

Начиная с .NET 5 в C# 9 можно использовать краткую форму записи new:

```
Point3D Point = new();
```

Если структура содержит только публичные поля (не путать со свойствами) и методы, то можно не вызывать конструктор, а сразу назначить значение полей и после этого вызывать методы структуру. Например:

```
public struct Point3D
{
public double X;
public double Y;
public double Z;
public override string ToString()
{
return $"({X},{Y},{Z})";
}};
      //не вызывается конструктор, а сразу задается значение
полей
Point3D Point;
Point.X = 100;
Point.Y = 100;
Point.Z = 100;
Console.WriteLine(Point.ToString());
```

Начиная с версии C# 10 полям структуры можно присваивать значения по умолчанию, однако, в этом случае необходимо будет вызвать new(), чтобы создать экземпляр структуры. Например:

```
public struct Point3D
{
public double X = 10;
public double Y = 5;
public double Z = 8;
public override string ToString()
{
return $"({X},{Y},{Z})";
}}
```

Попытка вывести в консоль значения полей используя код:

```
Point3D point;
Console.WriteLine(point.ToString());
```

будет вызывать ошибку «Попытка доступа к неинициализированной переменной». Поэтому, необходимо получать доступ к переменной point с использованием оператора new:

```
Point3D point = new();
Console.WriteLine(point.ToString());
```

3. Конструкторы структур

Любая структура имеет как минимум один конструктор (конструктор по умолчанию) без параметров. При этом, можно создавать свои конструкторы и точно также, как и с классами, вызывать их по цепочке, например:

```
public struct Point3D
{
public double X = 10;
public double Y = 5;
public double Z = 8;
public override string ToString()
{
return $"({X},{Y},{Z})";
}
public Point3D(double x) : this()
{
X = x;
Y = 0;
Z = 0;
}
public Point3D(double x, double y) : this(x)
{
Y = y;
Z = 0;
}
public Point3D(double x, double y, double z) : this(x, y)
{
Z = z;
}}
```

Создаем структуры (struct)

```
Point3D point1 = new(10);
Point3D point2 = new(10, 20);
Point3D point3 = new(10, 30, 40);
Console.WriteLine(point1);
Console.WriteLine(point2);
Console.WriteLine(point3);
```

Вывод консоли:

```
(10,0,0)
(10,20,0)
(10,30,40)
```

Начиная с версии C# 10 возможно создать для структуры свой конструктор без параметров:

```
      //конструктор без параметров
public Point3D()
{
X = 10;
Y = 20;
Z = 30;
}
```

4. Инициализатор структур struct

Значения полей и свойств структуры, как в случае и с классами, можно задавать непосредственно при создании, используя следующую языковую конструкцию:

```
Point3D Point4 = new() { X = 24, Y = 45, Z = 22 };
```

То есть вначале объявляется переменная, затем вызывается конструктор и затем в фигурных скобках указываются имена полей или свойств и их значения. Даже, если мы создадим и проинициализируем структуру вот так:

```
Point3D Point4 = new(10, 20, 30) { X = 24, Y = 45, Z = 22 };
Console.WriteLine(Point4);
```

то значения полей будут теми, которые указываем в инициализаторе, т.е. 24, 45 и 22.

Копирование структур с изменением значений (оператор with).

Начиная с версии C# 10 можно копировать значения структур с изменениями, например:

```
Point3D Point4 = new() { X = 24, Y = 45, Z = 22 };
Point3D Point5 = Point4 with { X = 100 };//меняем значение X
Console.WriteLine(Point4);
Console.WriteLine(Point5);
```

Вывод в консоли:

```
(24,45,22)
(100,45,22)
```

5. Отличие структуры от класса в C#

Структура – тип значений, класс – ссылочный тип

Основное отличие struct от class заключается в том, что структура храниться целиком в стеке, а объект класса храниться

в куче, а ссылка на него – в стеке. В результате этого, доступ к данным структуре будет пусть не намного, но быстрее, чем к классу. Структуры не поддерживают наследование

В отличие от классов C#, наследование структур не поддерживается, то есть вот такой код приведет к ошибке:

```
struct Point3DType2 : Point3D
{ }
```

6. Когда использовать структуры (struct), а когда классы (class) в C#

Основная рекомендация от Microsoft может быть сформулирована следующим образом: структуры (struct) стоит использовать в том случае, если ваш объект содержит минимальное количество каких-либо логически связанных операций или не содержит их вообще.

Например, использование структур вполне оправдано в примерах выше – описание точки в трехмерном пространстве. Максимум логики, которую можно добавить в структуру – это переопределить операторы сложения, вычитания и равенства.
Если же пробуем описать с помощью своего типа данных, например, автомобиль, то возможны различные варианты: проверка наличия топлива в баке, технические характеристики, оценка состояния в зависимости от каких-либо внешних или внутренних факторов и т.д. Соответственно, в этом случае, более предпочтительным будет использование не структуры, а класса.

Задания и порядок выполнения работы

Задание 1. Напишите программу, в которой объявляется перечисление для представления дней недели. Предложите методы (или добавьте в главный метод блоки кода), позволяющие по числовому значению определить день недели (с учетом периодической повторяемости дней), а также позволяющие по двум значениям из перечисления определить минимальное количество дней между соответствующими днями недели.

Задание 2. Создайте приложение, которое будет работать с различными видами продуктов и учитывать их количество на

складе. Для этого нужно использовать перечисление, чтобы определить все возможные виды продуктов.

Задание 3. Напишите программу для работы с цветами, которая будет позволять пользователю выбирать цвет из палитры. Для этого можно использовать перечисление со всеми возможными цветами.

Задание 4. Напишите программу, в которой объявляется структура с целочисленным, текстовым и символьным полями. Предложите такие версии конструктора: с тремя аргументами (целое число, текст, символ), с двумя аргументами (целое число и текст) и с одним аргументом (текст). В структуре должен быть метод, при вызове которого отображаются значения полей экземпляра структуры.

Задание 5. Напишите программу, содержащую структуру с целочисленным массивом. Опишите конструктор с одним аргументом, определяющим размер массива. Массив должен заполняться случайными числами. В структуре должны быть методы, возвращающие результатом наибольший элемент в массиве, наименьший элемент в массиве, а также метод, возвращающий среднее значение элементов в массиве (сумма элементов массива, деленная на количество элементов в массиве)

Контрольные вопросы

1. Что такое структуры в C# и зачем они нужны?
2. В чем отличие структур от классов?
3. Какие особенности имеют структуры в C# относительно других языков программирования?
4. Что такое перечислимый тип данных (enum) в C# и как его создать?
5. Какие преимущества использования перечислимых типов данных в сравнении с другими типами данных?
6. В чем особенность работы с значениями перечислимого типа данных в C# по сравнению с другими языками программирования?
7. В каких ситуациях использование структур и перечислимых типов является предпочтительным?

8. Можно ли наследовать от структур в C#? Если нет, то почему?

9. В чем заключается разница между модификаторами доступа public, private и protected при работе со структурами?

10. Какие особенности имеет инициализация переменных структурного типа данных и перечислимого типа?

Лабораторная работа № 25

Тема: **«Использование пользовательских структур для решения задач»**

Цель: научиться использовать пользовательские структуры для решения задач в C#, изучить их преимущества и недостатки, а также понять, в каких случаях их применение является наиболее оптимальным.

Краткие теоретические сведения

В C# структуры – это пользовательские типы данных, которые сочетают в себе свойства и классов, и примитивных типов. Они полезны для создания новых типов данных, которые имеют определенную логику работы и семантику. Структуры обычно используются для представления сложных объектов, состоящих из множества полей и методов. Они также используются для оптимизации производительности, особенно когда данные имеют небольшой размер и должны обрабатываться быстро.

Структуры создаются с помощью ключевого слова `struct`. Они имеют ряд отличий от классов:

– Структуры не могут иметь конструкторов с параметрами или без параметров.

– Все члены структуры по умолчанию являются публичными.

– Значения полей структуры по умолчанию равны своим значениям по умолчанию.

Структуры могут быть анонимными и могут использоваться в качестве возвращаемых типов методов и параметров делегатов. Также структуры могут быть расширены с помощью наследования, но для этого нужно указать ключевое слово `class`.

Пример структуры:

```
public struct Point
{
   public int X { get; set; }
   public int Y { get; set; }

   public Point(int x, int y)
   {
      X = x;
```

```
        Y = y;
    }

    public static Point operator +(Point p1, Point p2)
    {
        return new Point(p1.X + p2.X, p1.Y + p2.Y);
    }
}
```

В этом примере создается структура `Point`, которая представляет точку на плоскости. Структура имеет два свойства (`X` и `Y`) и конструктор, который инициализирует эти свойства. Также структура определяет оператор `+`, который позволяет складывать две точки.

Задания и порядок выполнения работы

Задание 1. Реализуйте структуру, представляющую почтовый адрес. Она должна содержать поля для индекса, города, страны, улицы и номера дома. Показать работу с этой структурой.

Задание 2. Реализуйте структуру для представления книги, содержащую поля для названия, автора, года издания, жанра и количества страниц. Показать работу с этой структурой.

Задание 3. Реализуйте структуру для хранения информации о сотруднике, включающую имя, должность, зарплату, отдел и дату найма. Показать работу с этой структурой.

Задание 4. Создайте структуру для представления записи в блоге, содержащую заголовок, текст, дату публикации, теги и количество комментариев.

Задание 5. Реализуйте структуру `Point`, содержащую координаты `x` и `y`. Создайте методы для вычисления расстояния между двумя точками и определения, находится ли точка внутри круга заданного радиуса и центра.

Задание 6. Напишите программу, в которой есть структура с двумя целочисленными полями. Предложите статический метод, аргументом которому передается целочисленный массив. Результатом метод возвращает экземпляр структуры, первое поле которого равно значению максимального (или минимального) элемента в массиве, а второе поле экземпляра равно индексу этого элемента в массиве

Контрольные вопросы

1. Каковы основные принципы работы со структурами в C#?
2. Что такое структура в C# и чем она отличается от других типов данных?
3. Как объявить и использовать структуру в коде на C#?
4. Можно ли в C# использовать структуры в качестве параметров методов и свойств?
5. Можно ли в структурах определять конструкторы и деструкторы?
6. Как происходит инициализация структур по умолчанию в C#?
7. Можно ли расширять структуры в C#, то есть добавлять новые поля или методы?
8. Что такое анонимные структуры в C# и в каких ситуациях они могут быть полезны?
9. Возможно ли использование структур в качестве параметров делегатов?
10. В чем заключаются преимущества и недостатки структур по сравнению с классами в C#?

Лабораторная работа № 26

Тема: «Разработка Windows Forms приложения Инженерный калькулятор»

Цель: разработать Windows Forms приложение «Инженерный калькулятор», которое будет выполнять основные математические операции (сложение, вычитание, умножение, деление, возведение в степень, извлечение корня) и дополнительные инженерные функции (тригонометрические функции, логарифмы, факториал), а также обрабатывать ошибки и предотвращать деление на ноль.

Краткие теоретические сведения

Статический класс Math содержит ряд методов и констант для выполнения математических, тригонометрических, логарифмических и других операций. Так как класс статический, то и все его методы также являются статическими, т.е. вызывать эти методы можно без создания объекта типа Math. Рассмотрим основные методы этого класса.

1. Тригонометрические функции (синус, косинус, тангенс)

Cos(x) – возвращает косинус указанного угла x. Например:

```
x = Math.Cos(1);//0,5403023058681398
x = Math.Cos(0.5); //0,8775825618903728
x = Math.Cos(0.75);//0,7316888688738209
x = Math.Cos(-1);//0,5403023058681398
```

Sin(x) – возвращает синус указанного угла x. Например:

```
x = Math.Sin(1);//0,8414709848078965
x = Math.Sin(0.5); //0,479425538604203
x = Math.Sin(0.75);//0,6816387600233341
x = Math.Sin(-1);//-0,8414709848078965
```

Tan(x) – возвращает тангенс указанного угла x. Например:

```
x = Math.Tan(1);//1,5574077246549023
x = Math.Tan(0.5); //0,5463024898437905
x = Math.Tan(0.75);//0,9315964599440725
x = Math.Tan(-1);//-1,5574077246549023
```

2. Обратные тригонометрические функции (арккосинус, арксинус, арктангенс).

Acos(x) – вычисляет арккосинус заданного числа. Параметр x должен находиться в диапазоне от -1 до 1. Значение, возвращаемое методом – радианы. Пример:

```
x = Math.Acos(1); //0
x = Math.Acos(0); //1,5707963267948966
x = Math.Acos(-1);//3,141592653589793
x = Math.Acos(0.5);//1,0471975511965979
```

Asin(x) – вычисляет арксинус числа. Параметр x должен находиться в диапазоне от -1 до 1. Например:

```
x = Math.Asin(1); //1,5707963267948966
x = Math.Asin(0); // 0
x = Math.Asin(10);//ошибка x>1
x = Math.Asin(-1);//-1,5707963267948966
x = Math.Asin(0.6);//0,6435011087932844
```

Atan(x) – возвращает арктангенс числа. Значение, возвращаемое методом, лежит в диапазоне от -Пи/2 до +Пи/2. Например:

```
x = Math.Atan(1); //0,7853981633974483
x = Math.Atan(0); //0
x = Math.Atan(10);//1,4711276743037347
x = Math.Atan(-1);//-0,7853981633974483
x = Math.Atan(0.6);//0,5404195002705842
```

Atan2(x, y) – возвращает арктангенс для точки в декартовой системе координат с координатами (x,y). Например:

```
x = Math.Atan2(1,1); //0,7853981633974483
x = Math.Atan2(1,-1); //2,356194490192345
x = Math.Atan2(-1,1);//-0,7853981633974483
x = Math.Atan2(-1,-1);//-2,356194490192345
```

Этот метод возвращает значение (ϑ), в зависимости от того, в каком квадранте располагается точка, а именно:

для (x, y) в квадранте 1: $0<\vartheta<\pi/2$.

для (x, y) в квадранте 2: $\pi/2<\vartheta\leq\pi$.

для (x, y) в квадранте 3: $-\pi<\vartheta<-\pi/2$.

для (x, y) в квадранте 4: $-\pi/2<\vartheta<0$.

При этом, стоит особо отметить, что точки могут лежать и за пределами указанных квадрантов, например, когда одна из

координат равна нулю. В этом случае метод работает следующим образом:

если y равно 0, и x не является отрицательным, то ϑ=0.

если y равно 0, и x является отрицательным, то $\vartheta=\pi$.

если y – положительное число, а x равно 0, то $\vartheta=\pi/2$.

если y – отрицательное число, а x равно 0, то $\vartheta=-\pi/2$.

если y равен 0, и x равен 0, то ϑ=0.

3. Гиперболические функции (гиперболический косинус, гиперболический синус, гиперболический тангенс)

Cosh(x) – вычисляет гиперболический косинус указанного угла

```
x = Math.Cosh(1);//1,5430806348152437
x = Math.Cosh(0.5); //1,1276259652063807
x = Math.Cosh(0.75);//1,2946832846768448
x = Math.Cosh(-1);//1,5430806348152437
```

Sinh(x) – вычисляет гиперболический синус указанного угла

```
x = Math.Sinh(1);//1,1752011936438014
x = Math.Sinh(0.5); //0,5210953054937474
x = Math.Sinh(0.75);//0,82231673193583
x = Math.Sinh(-1);//-1,1752011936438014
```

Tanh(x) – вычисляет гиперболический тангенс указанного угла

```
x = Math.Tanh(1);//0,7615941559557649
x = Math.Tanh(0.5); //0,46211715726000974
x = Math.Tanh(0.75);//0,6351489523872873
x = Math.Tanh(-1);//-0,7615941559557649
```

4. Обратные гиперболические функции (ареакосинус, ареасинус, ареатангенс)

Acosh(x) – вычисляет ареакосинус. Параметр x должен быть больше 1. Например:

```
x = Math.Acosh(1); //0
x = Math.Acosh(0); // ошибка - x<1
x = Math.Acosh(10);//2,993222846126381
x = Math.Acosh(3.14);//1,810991348900196
x = Math.Acosh(10000);//9,903487550036129
```

Asinh(x) – вычисляет ареасинус. Например,

```
x = Math.Asinh(1);//0,881373587019543
x = Math.Asinh(0.5); //0,48121182505960347
x = Math.Asinh(0.75);//0,6931471805599453
```

```
x = Math.Asinh(-1);//-0,881373587019543
```

Atanh(x) – вычисляет ареатангенс. Значение x должно быть в пределах от -1 до 1. Например,

```
x = Math.Atanh(-0.5);//-0,5493061443340549
x = Math.Atanh(0.5); //0,5493061443340549
x = Math.Atanh(0.75);//0,9729550745276566
x = Math.Atanh(-0.75);//-0,9729550745276566
```

5. Вычисление логарифмов в C

ILogB(x) – вычисляет целочисленный логарифм с основанием 2 указанного числа, то есть значение (int)log2(x).

```
x = Math.ILogB(15);//3
```

Log2(x) – вычисляет логарифм с основанием 2 указанного числа.

```
x = Math.Log2(15);//3,9068905956085187
```

Log(x) – вычисляет натуральный логарифм (с основанием e) указанного числа.

```
x = Math.Log(15);//2,70805020110221
```

Log(x, y) – вычисляет логарифм указанного числа x по основанию y.

```
x = Math.Log(15, 2);//3,9068905956085187
```

Log10(x) – вычисляет логарифм с основанием 10 указанного числа.

```
x = Math.Log10(15);//1,1760912590556813
```

6. Методы округления чисел в C

Ceiling(x) – возвращает наименьшее целое число, которое больше или равно заданному числу.

```
x = Math.Ceiling(7.256);//8
```

Floor(x) – возвращает наибольшее целое число, которое меньше или равно указанному числу.

```
x = Math.Floor(7.256);//7
```

Round(x) – округляет значение до ближайшего целого значения; значения посередине округляются до ближайшего четного числа.

```
x = Math.Round(7.256);//7
```

Round(x, Int32 y) – округляет значение до указанного числа знаков после запятой; значения посередине округляются до ближайшего четного числа.

```
x = Math.Round(7.256, 1);//7,3
```

Round(x, Int32 y, MidpointRounding) – округляет значение до указанного числа дробных цифр, используя указанное соглашение о округлении. В таблице 26.1 представлено сравнение различных методов округления

```
x = Math.Round(7.256, 2, MidpointRounding.ToZero); //7,25
x = Math.Round(7.256, 2, MidpointRounding.ToPositiveInfinity);
//7,26
x = Math.Round(7.256, 2, MidpointRounding.ToNegativeInfinity);
//7,25
x = Math.Round(7.256, 2, MidpointRounding.ToEven); //7,26
```

Round(x, MidpointRounding) – округляет значение на целое число, используя указанное соглашение об округлении.

```
x = Math.Round(7.256, MidpointRounding.ToZero);//7
```

Truncate(x) – вычисляет целую часть заданного числа.

```
x = Math.Truncate(7.256);//7
```

Таблица 26.1. Сравнение различных методов округления в C#

x	7,256	7,556	7,5
Ceiling(x)	8	8	8
Floor(x)	7	7	7
Round(x)	7	8	8
Round(x, 2)	7,26	7,56	7,5
Round(x, ToZero)	7	7	7
Round(x, 2, ToZero)	7,25	7,55	7,5
Round(x, 2, ToPositiveInfinity)	7,26	7,56	7,5
Round(x, 2, ToNegativeInfinity)	7,25	7,55	7,5
Round(x, 2, ToEven)	7,26	7,56	7,5
Truncate(x)	7	7	7

7. Возведение в степень и извлечение корней в C#

Cbrt(x) – возвращает кубический корень из x.

```
x = Math.Cbrt(27);//3
```

Exp(x) – возвращает e, возведенное в указанную степень x.

```
x = Math.Exp(3);//20,085536923187668
```

Pow(x, y) – возвращает число x, возведенное в степень y.

```
x = Math.Pow(3,3);//27
```

Sqrt(x) – возвращает квадратный корень из числа x.

```
x = Math.Sqrt(9);//3
```

8. Прочие математические операции

Abs(x) – возвращает абсолютное значение числа.

```
x = Math.Abs(-2.7);//2.7
x = Math.Abs(2.7);//2.7
```

BigMul(Int32, Int32) – умножает два 32-битовых числа и возвращает значение в Int64.

```
double x = Math.BigMul(Int32.MaxValue,
Int32.MaxValue);//4,6116860141324206E+18
```

Если в программе предполагается использование очень больших чисел, то стоит обратить внимание на тип BigInteger

BitDecrement(x) – возвращает ближайшее самое маленькое значение, которое меньше, чем x.

```
x = Math.BitDecrement(1);//0,9999999999999999
x = Math.BitDecrement(-1);//-1,0000000000000002
```

BitIncrement(Double) – возвращает наименьшее ближайшее значение, которое больше указанного значения.

```
x = Math.BitIncrement(1);//1,0000000000000002
x = Math.BitIncrement(-1);//-0,9999999999999999
```

Clamp(x, min, max) – Возвращает x, ограниченное диапазоном от min до max включительно.

```
x = Math.Clamp(3, 5, 10);//5
x = Math.Clamp(4, 5, 10);//5
x = Math.Clamp(11, 5, 10);//10
```

CopySign(Double, Double) – возвращает значение с величиной x и знаком y.

```
double x = Math.CopySign(-3, 5);//3 (знак + взят у 5)
x = Math.CopySign(5, -10);//-5 (знак - взят у -10)
```

DivRem(Int32, Int32, Int32) – Вычисляет частное чисел и возвращает остаток в выходном параметре.

```
x = Math.DivRem(-3, 5, out int y);//x = 0; y = -3
x = Math.DivRem(10, 10, out int z);//x = 1; z = 0
```

FusedMultiplyAdd(x, y, z) – возвращает значение (x * y) + z, округленное в рамках одной тернарной операции.

```
x = Math.FusedMultiplyAdd(10.4566d, 5.56012f,
10.83789f);//68,97784156904221
x = (10.4566d * 5.56012f) + 10.83789f;//68,9778415690422
```

Метод FusedMultiplyAdd округляет значение один раз – в конце вычислений, в то время как обычное вычисление (x*y)+z вначале округляет значение, полученное в скобках, затем

добавляет к полученному значению z и ещё раз округляет результат.

IEEERemainder(x, y) – Возвращает остаток от деления одного указанного числа на другое указанное число.

```
double x = Math.IEEERemainder(10, 3);//1
```

Функция IEEERemainder не совпадает с оператором %, который также вычисляет остаток от деления. Различие заключается в формулах, используемых при вычислениях. Оператор действует следующим образом:

```
a % b =  (Math.Abs(a) - (Math.Abs(b) * (Math.Floor(Math.Abs(a)
/ Math.Abs(b))))) * Math.Sign(a)
```

В то время, как метод IEEERemainder производит вычисления следующим образом:

```
IEEERemainder(a,b) = a-(b * Math.Round(a/b))
x = Math.IEEERemainder(-16.3, 4.1);//0,09999999999999787
x = -16.3 % 4.1;//-4,000000000000002
```

Max(x, y) – возвращает большее из чисел.

```
x = Math.Max(-16.3, 4.1);//4,1
x = Math.Max(10, 5);//10
```

MaxMagnitude(x, y) – возвращает большую величину из двух чисел двойной точности с плавающей запятой. При сравнении двух чисел не учитывается знак.

```
x = Math.MaxMagnitude(-16.3, 4.1);//-16,3
x = Math.MaxMagnitude(10, 5);//10
```

Min(Byte, Byte) – возвращает меньшее из двух чисел.

```
x = Math.Min(-16.3, 4.1);//-16,3
x = Math.Min(10, 5);//5
```

MinMagnitude(x, y) – возвращает меньшую величину из чисел. При сравнении не учитывается знак.

```
x = Math.MinMagnitude(-16.3, 4.1);//4.1
x = Math.MinMagnitude(10, -5);//-5
```

ScaleB(x, y) – Возвращает значение x * 2^n, вычисленное наиболее эффективно.

```
x = Math.ScaleB(-16.3, 8);//4172,8
```

Sign(x) – возвращает целое число, указывающее знак десятичного числа.

```
x = Math.Sign(-16.3);//-1
x = Math.Sign(10);//1
x = Math.Sign(0);//0
```

9. Константы класса Math

В классе Math также определены две константы – E и PI:

```
x = Math.E;//2,718281828459045
x = Math.PI;//3,141592653589793
```

10. Ввод и вывод информации с помощью диалоговых окон

Для ввода данных или вывода сообщений можно использовать диалоговые окна. Рассмотрим пример простейшего приложения для вычисления значения sin(x), в котором для ввода и вывода информации используются диалоговые окна. Для использования диалогового окна для ввода данных необходимо подключить пространство имен Microsoft.VisualBasic.

```
using Microsoft.VisualBasic;
using System.Runtime.CompilerServices;
namespace WinFormsApp4
{

    public partial class Form1 : Form
    {

        public class calculator
        {
        public double getresult(double x) { return Math.Sin(x);
}
        }
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            calculator mycalc= new calculator();

            string input = Interaction.InputBox("Введите
значение x", "Ввод данных. Как разделитель используется знак
запятой");
            double z=Convert.ToDouble(input);
            MessageBox.Show(mycalc.getresult(z).ToString(),
"Результат вычислений!");
        }
    }
}
```

В данном примере, был создан класс calculator, который содержит всего один метод для вычисления значения sin(x). После

нажатия на кнопку button1 происходит вызов диалогового окна, в котором пользователю предлагается ввести значение x. Полученное значение преобразовывается к типу double и передается в метод getresult объекта mycalc, который представляет собой экземпляр класса calculator.

Задания и порядок выполнения работы

Задание 1. Разработайте Windows Forms приложение, которое реализует математические функции согласно своему варианту. Номер варианта определяется путем взятия остатка от деления на 3 своего номера по списку в группе.

Вариант 1. Тригонометрические, обратные тригонометрические, возведение в степень.

Вариант 2. Извлечение корней, округление различными способами, вычисления с логарифмами.

Вариант 3. Гиперболические, обратные гиперболические, тригонометрические.

Задание 2. Добавить в приложение кнопку, обработчик нажатия которой выполняет вычисление первых 1000 простых чисел используя алгоритм нахождения простых чисел (решето Эратосфена). Результаты вычислений должны сохраняться в файл числа.txt.

Задание 3. Добавить в приложение кнопку, обработчик нажатия которой выполняет вычисление факториала, заданного пользователем числа.

Задание 4. Добавить в приложение возможности вызова исключений, в случае некорректного ввода данных пользователем.

Контрольные вопросы

1. Что такое статические методы и зачем они нужны в C#?

2. Какие математические функции предоставляет класс Math в C#?

3. Как использовать класс Math для выполнения математических операций?

4. Что означает, если метод в классе Math имеет модификатор static?

5. Какие исключения могут возникнуть при использовании класса Math и как их обрабатывать?

6. Какие другие стандартные библиотеки доступны в C# для выполнения математических операций?

7. Как использовать класс MathF для выполнения операций с плавающей запятой?

8. Чем отличается класс Math от других библиотек для выполнения математических операций в C#?

9. Какие ограничения есть у класса Math при выполнении некоторых математических операций, таких как извлечение корня из отрицательного числа?

10. Приведите пример использования класса Math для решения математической задачи.

Заключение

В данной части лабораторного практикума были представлены лабораторные работы, направленные на дальнейшее изучение синтаксиса языка C#.

Вторая часть была посвящена разработке приложений с графическим интерфейсом на Windows. Были рассмотрены создание и настройка форм, работа с файлами, обработка событий. Также в лабораторных работах изучались абстрактные классы, интерфейсы, делегаты, а также различные элементы управления, такие как кнопки, списки, метки и поля ввода.

Выполнение лабораторных работ данного практикума позволит получить навыки для создания профессиональных графических приложений. Кроме того, студенты научатся использовать возможности регулярных выражений, а также знакомятся с новыми возможностями языка C#.

Выполнив все лабораторные работы данного практикума, студенты должны получить четкое понимание основных принципов разработки Windows Forms приложений, научиться правильно использовать ключевые элементы управления, а также приобрести навыки решения задач программирования, связанных с разработкой графических приложений.

Авторы надеются, что данный лабораторный практикум станет полезным инструментом в обучении студентов основам программирования на платформе .Net, и желают успехов в освоении этого увлекательного и перспективного направления информационных технологий.

Список рекомендуемой литературы

1. Васильев, А.Н. Программирование на C# для начинающих. Особенности языка / А.Н. Васильев. – Москва : Эксмо, 2019. – 528 с.

2. Гриффитс, И. Программируем на C# 8.0. Разработка приложений / И. Гриффитс, – СПб : Питер, 2021. – 944 с.

3. Прайс, М. C# 10 и .NET 6. Современная кросс-платформенная разработка / М. Прайс. – СПб.: Питер, 2023. – 848 с.

4. Танвар, Ш. Параллельное программирование на C# и .NET Core / Ш. Танвар, пер. с англ. А. Д. Ворониной; ред. В.Н. Черников. – М.: ДМК Пресс, 2021. – 272 с.: ил.

5. Троелсен, Э. Язык программирования C# 7 и платформы .NET и .NET Core 8-е изд. / Э. Троелсен, Ф. Джепикс, : Пер. с англ. – СПб. : ООО “Диалектика”, 2018 – 1328 с.

Учебное издание

ШВЫРОВ Вячеслав Владимирович
КАПУСТИН Денис Алексеевич
ШИШЛАКОВА Виктория Николаевна

ПРОГРАММИРОВАНИЕ ДЛЯ ПЛАТФОРМЫ .NET
ЧАСТЬ 2. ПРИЛОЖЕНИЯ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Лабораторный практикум

В авторской редакции
Редактор – Швыров В. В.
Дизайн обложки – Швыров В. В.
Корректор – Шишлакова В. Н.
Верстка – Швыров В. В.

Подписано в печать 06.06.2024. Бумага офсетная.
Гарнитура Times New Roman.
Печать ризографическая. Формат 60×84/16. Усл. печ. л. 6,74.
Тираж 50 экз. Заказ № 46

Издательство ЛГПУ
ФГБОУ ВО «ЛГПУ»
ул. Оборонная, 2, г. Луганск, ЛНР, 291011. Т/ф: +7-857-258-03-20
e-mail: knitaizd@mail.ru