Федеральное государственное автономное образовательное
учреждение высшего образования

«КРЫМСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
имени В.И. ВЕРНАДСКОГО»

кафедра компьютерной инженерии и моделирования

Практикум по учебной дисциплине

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Направления подготовки:
09.03.01 "Информатика и вычислительная техника",
09.03.04 "Программная инженерия"

Квалификационный уровень - бакалавр

Симферополь, 2024

В.В.Милюков. Практикум по учебной дисциплине «Объектно-ориентированное программирование» . – Симферополь: ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского», 2024. – 63 с.

Практикум по учебной дисциплине «Объектно-ориентированное программирование» содержат ссылки на литературу для предварительного изучения, задания для практических работ, методические указания к ним (к некоторым работам могут выдаваться исходные коды незаконченных проектов).

Утверждено на заседании кафедры компьютерной инженерии и моделирования, протокол от 2 сентября 2024г. № 1

Заведующий кафедрой компьютерной
инженерии и моделирования _____ (Милюков В.В.)

Издается по решению Методического совета Физико-технического института ФГАОУ ВО «КФУ им. В.И. Вернадского»
(протокол № от 26.09.2023 г.).

Председатель методической комиссии _____ (М.А.Бакуменко)

Практическая работа №1.[1]

Тема: Использование программных конструкций C#

Цель работы: Научиться создавать простейшие консольные и WPF приложения на языке C# в среде Visual Studio, изучить возможность создания самодокументируемых приложений.

Научиться преобразовывать различные типы данных в C#, познакомиться с типом данных Decimal, научиться грамотно использовать циклы для итерационных вычислений с контролем погрешности, обрабатывать события нажатия клавиш, научиться использовать классы String, StringBuilder, научиться создавать самодокументируемые XML справочные файлы.

Ключевые понятия: интегрированная среда разработки IDE (Integrated Development Envirionment) Visual Studio, общеязыковая исполнительная среда CLR, библиотека классов FCL, общеязыковые спецификации CLS, решение (solution), проект (project), пространство имен (namespace), сборка (assembly), MSIL (Microsoft Intermediate Language, IL), управляемый код, двухэтапная компиляция, дизассемблер, обозреватель решений (Solution Explorer), Windows.Forms и WPF проекты, класс, статический тип, динамический тип, встроенные типы, типы-значения, ссылочные типы, фундаментальные типы: логический, символьный, целый, с плавающей точкой, void, указатели, ссылки, массивы, перечисления, структуры, классы, преобразования типов: упаковка Boxing, неявное преобразование, явное преобразование, класс Convert.

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентацию лектора курса: «Введение в C# и платформу .NET" и «Использование конструкций языка C#» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Материалы сайта Metanit.com.
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка.[2] (Приложение A, Комментарии к документации).

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы
3. Программный код для каждого из заданий с подробными комментариями

Задание 1: Приведение и преобразование типов

Написать программу преобразования типов, в которой предусмотреть:

1) Неявное преобразование простых и ссылочных типов, в виде комментариев внести в программу таблицу неявных преобразований;

2) Явное преобразование простых и ссылочных типов, в виде комментариев внести в программу таблицу явных преобразований;

3) Безопасное приведение ссылочных типов с помощью операторов as и is;

4) Пользовательское преобразование типов Implicit, Explicit;

5) Преобразование с помощью класса Convert и преобразование строки в число с помощью методов Parse, TryParse.

6) Написать программу, позволяющую ввод в текстовое поле TextBox только символов, задающих правильный формат вещественного числа со знаком.

---

[1] В практических работах частично использованы официальные материалы Microsoft

[2] Файл документации «Корпорация Microsoft. C#. Спецификация языка» входит в комплект поставки Visual Studio и находится в каталоге Program Files\Microsoft Visual Studio ***\VC#\Specification\1049.

Углубленное задание (5 бонусных баллов к 100 балльной оценке за первую работу).

Написать программу, в которой реализуется явное и неявное преобразование любого типа в любой. Для выбора вариантов можно ограничиться типами: char, string, byte, int, float, double, decimal, bool, object. Указание - использовать тип dynamic с динамической типизацией.

Задание 2: Вычислить квадратный корень с контролем точности

In this exercise, you will write a program that prompts the user for a numeric value and then uses Newton's method to calculate the square root of this number. You will display the result, and compare it to the **double** value that is calculated by using the **Math.Sqrt** method in the .NET Framework class library.

Scenario

Some of the software that is being developed to support devices that perform scientific analysis requires applications to perform calculations with a high degree of accuracy. The .NET Framework uses the **double** type to perform many of its calculations. The **double** type has a very large range, but the accuracy is not always sufficient. The **decimal** type provides a higher degree of accuracy at the cost of a smaller range and increased memory requirements. However, this accuracy is important. One scientific calculation requires the ability to calculate square roots to a high degree of accuracy. You decide to implement Newton's algorithm for estimating and successively refining square roots, but generate the result by using the **decimal** type.

The process that Newton used for calculating the square root of 10 is as follows:

1. Start with an initial guess: use the value that you want to find the square root of and divide by 2.

In this case, 10 / 2, has the value 5.

2. Refine the guess by dividing the original number by the previous guess, adding the value of the previous guess, and dividing the entire result by 2: calculate ((number / guess) + guess) / 2.

In this example, calculate ((10 / 5) + 5) / 2 = 3.5

The answer 3.5 then becomes the next guess.

3. Perform the calculation ((number / guess) + guess) / 2 again, with the new guess. In this example, calculate ((10 / 3.5) + 3.5) / 2 = 3.17857

3.17857 is then the next guess.

4. Repeat this process until the difference between subsequent guesses is less than some predetermined amount. The final guess is the square root of 10 to the accuracy that was specified by this predetermined amount.

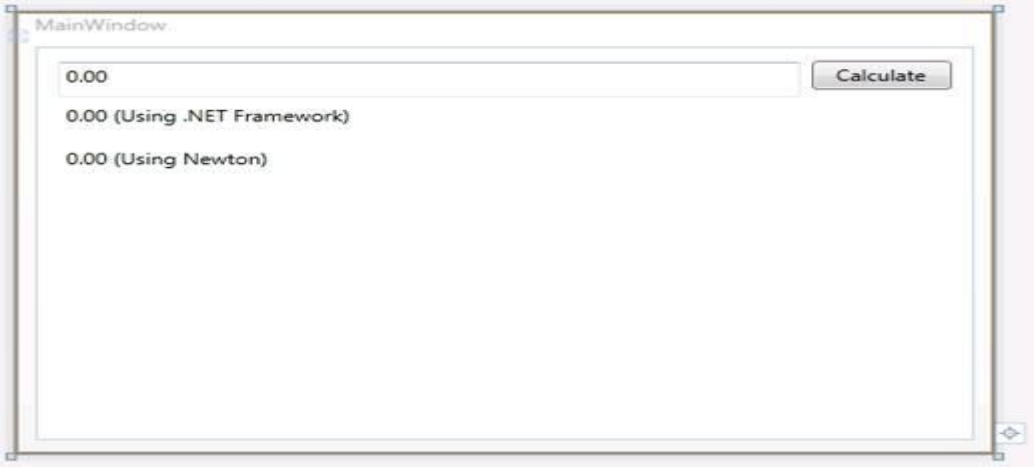
The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Calculate square roots by using the **Math.Sqrt** method of the .NET Framework.
4. Calculate square roots by using Newton's method.
5. Test the application.

- Task 1: Create a new WPF Application project
- Task 2: Create the user interface

1. Add **TextBox**, **Button**, and two **Label** controls to the MainWindow window. Place them anywhere in the window.
2. Using the Properties window, set the properties of each control. Leave any other properties at their default values.

The MainWindow window should look like the following screen shot.



- Task 3: Calculate square roots by using the Math.Sqrt method of the .NET Framework

1. Create an event handler for the **Click** event of the button.
2. In the **calculateButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into a **double** value. Store the **double** value in a variable called numberDouble. Use the **TryParse** method of the **double** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "Please enter a double," and then execute a **return** statement to quit the method.

Note: You can display a message in a message box by using the **MessageBox.Show** method.

3. Check that the value that the user enters is a positive number. If it is not, display a message box with the text "Please enter a positive number," and then return from the method.
4. Calculate the square root of the value in the numberDouble variable by using the **Math.Sqrt** method. Store the result in a double variable called squareRoot.
5. Format the value in the squareRoot variable by using the layout shown in the following code example, and then display it in the **frameWorkLabel Label** control.

```
99.999 (Using the .NET Framework)
```

Use the **string.Format** method to format the result. Set the **Content** property of a **Label** control to display the formatted result.

6. Build and run the application to test your code. Use the test values that are shown in the following table, and then verify that the correct square roots are calculated and displayed (ignore the "Using Newton" label for the purposes of this test).

Test value	Expected result
25	5
625	25
0.00000001	0.0001
–10	Message box appears with the message "Please enter a positive number"
Fred	Message box appears with the message "Please enter a double"
10	3.16227766016838
8.8	2.96647939483827

2.0	1.4142135623731
2	1.4142135623731

7. Close the application and return to Visual Studio.

- Task 4: Calculate square roots by using Newton's method

1. In the **calculateButton_Click** method, after the code that you added in the previous task, create a decimal variable called numberDecimal. Initialize this variable with the data that the user enters in the **inputTextBox TextBox** control, but convert it into a **decimal** this time (previously, you read it as a **double**). If the text that the user enters is not valid, display a message box with the text "Please enter a decimal," and then execute a **return** statement to quit the method.

Note: This step is necessary because the **decimal** and **double** types have different ranges. A number that the user enters that is a valid **double** might be out of range for the **decimal** type.

2. Declare a decimal variable called delta, and initialize it to the value of the expression **Math.Pow(10, −28)**. This is the smallest value that the **decimal** type supports, and you will use this value to determine when the answer that is generated by using Newton's method is sufficiently accurate. When the difference between two successive estimates is less than this value, you will stop.

Note: The **Math.Pow** method returns a double. You will need to use the **Convert.ToDecimal** method to convert this value to a decimal before you assign it to the delta variable.

3. Declare another decimal variable called guess, and initialize it with the initial guess at the square root. This initial guess should be the result of dividing the value in **numberDecimal** by 2.

4. Declare another decimal variable called result. You will use this variable to generate values for each iteration of the algorithm, based on the value from the previous iteration. Initialize the result variable to the value for the first iteration by using the expression **((numberDecimal / guess) + guess) / 2**.

5. Add a **while** loop to generate further refined guesses. The body of the **while** loop should assign result to guess, and generate a new value for result by using the expression **((numberDecimal / guess) + guess) / 2**. The **while** loop should terminate when the difference between result and guess is less than or equal to delta.

Note: Use the **Math.Abs** method to calculate the absolute value of the difference between result and guess. Using Newton's algorithm, it is possible for the difference between the two variables to alternate between positive and negative values as it diminishes. Consequently, if you do not use the **Math.Abs** method, the algorithm might terminate early with an inaccurate result.

6. When the **while** loop has terminated, format and display the value in the result variable in the **newtonLabel Label** control. Format the data in a similar manner to the previous task.

- Task 5: Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the correct square roots are calculated and displayed. Compare the value in the two labels, and then verify that the square roots that are calculated by using Newton's method are more accurate than those calculated by using the **Math.Sqrt** method.

Test value	.NET Framework	Newton's algorithm
25	5	5.000000000000000000000000000

625	25	25.000000000000000000000000000
0.00000001	0.0001	0.000100000000000000000000000000
10	3.16227766016838	3.162277660168379331998893544
8.8	2.96647939483827	2.966479394838265179484558976
2.0	1.4142135623731	1.414213562373095048801688724
2	1.4142135623731	1.414213562373095048801688724

2. As a final test, try the value 0.0000000000000000000000000001 (27 zeroes after the decimal point). Can you explain the result?

Обязательное дополнение. Предусмотреть вывод результатов (значения корня и погрешности) после каждой итерации, реализуемой при нажатии какой-нибудь клавиши или кнопки на форме. Указание: при использовании типа приложения WinForm свойство формы KeyPreview установить в True. В этом случае форма всегда будет реагировать на события клавиатуры, причем это будет происходить до реакции дочернего компонента, находящегося в фокусе.

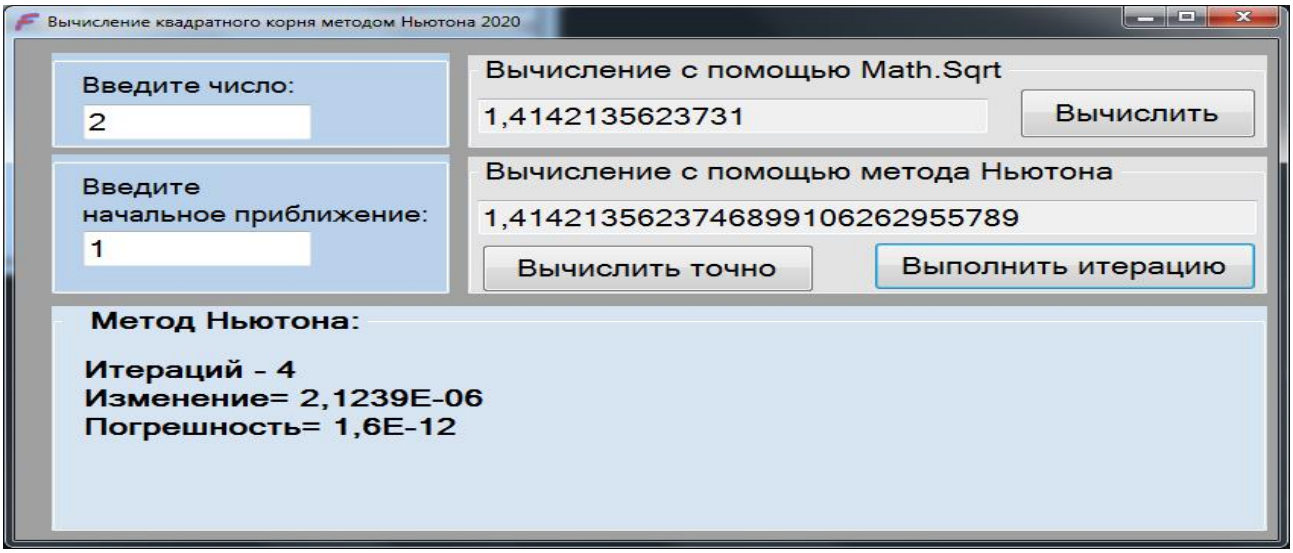


Рис. Образец интерфейса программы вычисления квадратного корня методом Ньютона

Углубленное задание (5 бонусных баллов). Разработать эффективный алгоритм выбора начального приближения для квадратного корня.
Например, если число y, из которого вычисляем корень, больше единицы, вычисляем *D* число цифр числа y слева от десятичной запятой, или если y < 1, вычисляем *D* число нулей, идущих подряд, справа от десятичной запятой, взятое со знаком минус. Тогда грубая оценка выглядит так: если y нечётно, $D = 2n + 1$, тогда используем приближение для корня $2 \cdot 10^n$, если *D* чётно, $D = 2n + 2$, тогда используем $6 \cdot 10^n$. При работе в двоичной системе, следует использовать оценку $2 \cdot 10^{(D/2)}$ (*D* - число двоичных цифр).

Задание 3: Конвертация десятичных целочисленных данных в любую систему исчисления вплоть до шестнадцатиричной

3.1. Конвертация десятичных целочисленных данных в двоичную систему. In this exercise, you will create another application that enables the user to enter an integer value, generate a string that holds the binary representation of this value, and then display the result.

Scenario

Another device has the requirement to display decimal numeric data in a binary format. You

have been asked to develop some code that can convert a non-negative decimal integer value into a string that contains the binary representation of this value.

The process for converting the decimal value 6 into its binary representation is as follows:

1. Divide the integer by 2, save the integer result, and use the remainder as the first binary digit.

In this example, 6 / 2 is 3 remainder 0. Save the character "0" as the first character of the binary representation.

2. Divide the result of the previous division by 2, save the result, and use the remainder as the next binary digit.

In this example, 3 / 2 is 1 remainder 1. Save the character "1" as the next character of the binary representation.

3. Repeat the process until the result of the division is zero.

In this example, 1 / 2 is zero remainder 1. Save the character "1" as the final character of the binary representation.

4. Display the characters saved in reverse order.

In this example, the characters were generated in the sequence "0", "1", 1", so display them in the order "1", "1", "0". The value 110 is the binary representation of the decimal value 6.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Add code to generate the binary representation of an integer value.
4. Test the application.

- Task 1: Create a new WPF Application project

Create a new project called IntegerToBinary by using the WPF Application.

- Task 2: Create the user interface

1. Add a **TextBox**, **Button**, and **Label** control to the MainWindow window. Place them anywhere in the window.
2. Using the Properties window, set the properties of each control. Leave any other properties at their default values.

The MainWindow window should look like the following screen shot.

- Task 3: Add code to generate the binary representation of an integer value

1. Create an event handler for the **Click** event of the button.
2. In the **convertButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into an **int** type. Store the integer value in a variable called i. Use the **TryParse** method of the **int** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "TextBox does not contain an integer," and then execute a **return** statement to quit the method.
3. Check that the value that the user enters is not a negative number (the integer-to-binary conversion algorithm does not work for negative numbers). If it is negative, display a message box with the text "Please enter a positive number or zero," and then return from the method.

4. Declare an integer variable called remainder and initialize it to zero. You will use this variable to hold the remainder after dividing i by 2 during each iteration of the algorithm.

5. Declare a StringBuilder variable called binary and instantiate it. You will use this variable to construct the string of bits that represent i as a binary value.

6. Add a **do** loop that performs the following tasks:

a. Calculate the remainder after dividing i by 2, and then store this value in the remainder variable.

b. Divide i by 2.

c. Prefix the value of remainder to the start of the string being constructed by the binary variable.

Terminate the **do** loop when i is less than or equal to zero.

Note: To prefix data into a **StringBuilder** object, use the **Insert** method of the **StringBuilder** class, and then insert the value of the data at position 0.

7. Display the value in the binary variable in the **binaryLabel Label** control.

Note: Use the **ToString** method to retrieve the string that a **StringBuilder** object constructs. Set the **Content** property of the **Label** control to display this string.

- Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the binary representations are generated and displayed.

Test value	Expected result
0	0
1	1
–1	Message box appears with the message "Please enter a positive number or zero"
10.5	Message box appears with the message "TextBox does not contain an integer"
Fred	Message box appears with the message "TextBox does not contain an integer"
4	100
999	1111100111
65535	1111111111111111
65536	10000000000000000

3.2. <u>Обязательное дополнение.</u>

Написать программу, реализующую перевод десятичного целого числа в любую другую систему счисления, вплоть до 16-ричной.

Рекомендуем использовать словать 16-ричных символов в виде строки String.

3.3. <u>Обязательное дополнение.</u>

Написать программу, реализующую перевод десятичного целого числа из арабской системы исчисления в римскую и обратно.

Рекомендуем использовать таблицу соответствия: (1,"I"), (4,"IV"), (5,"V"), (9,"IV"), (10,"X"), (40,"XL"), (50,"L"), (90,"XC"), (100,"C"), (400,"CD"), (500,"D"), (900,"CM"), (1000,"M").

<u>Задание 4: Создание документации для приложения</u>

In this exercise, you will add XML comments to your application, and use the Sandcastle tool to generate documentation for the application.

Scenario

You must ensure that your application is fully documented so that it can be maintained easily. You decide to add XML comments to the methods that you have added to the WPF application, and generate a help file.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add XML comments to the application.
3. Generate an XML comments file.
4. Generate a .chm file.

- Task 2: Add XML comments to the application

1. Display the MainWindow.xaml.cs file.
2. Add the XML comment in the following code example before the **MainWindow** class declaration.

```
/// <summary>
/// WPF application to read and format data
/// </summary>
```

3. Add the XML comment in the following code example before the **MainWindow** constructor.

```
/// <summary>
/// Constructor for MainWindow
/// </summary>
```

4. Add the XML comment in the following code example before the **testButton_Click** method.

```
/// <summary>
/// Read a line of data entered by the user.
/// Format the data and display the results in the
/// formattedText TextBlock control.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

5. Add the XML comment in the following code example before the **Windows_Loaded** method.

```
/// <summary>
/// After the Window has loaded, read data from the standard input.
/// Format each line and display the results in the
/// formattedText TextBlock control.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

6. Save MainWindow.xaml.cs.

Task 3: Generate an XML comments file

1. Set the project properties to generate an XML documentation file when the project is built.
2. Build the solution, and then correct any errors.
3. Verify that an XML comments file called comments.xml has been generated

Внимание. Для автоматического создания файла справки нужно в свойствах проекта на вкладке «построение» поставить галочку в чек-боксе XML –файл документации.

Практическая работа №2

Тема: Описание и вызов методов

Цель работы: Изучить на практике использование перегрузки и переопределения (Override) методов, статические и виртуальные методы, научиться передавать в методы простые типы по ссылке, передавать и возвращать из методов несколько значений, в том числе и неопределенное значение параметров. Научиться использовать компоненты Grid или DataGridView в Windows Forms или WPF приложениях.

Ключевые понятия: перегрузка, переопределение (Override) и скрытие методов, закрытые и открытые методы, статические и виртуальные методы, кортежи, Params.

Перед выполнением лабораторной работы рекомендуем:

8. Изучить презентацию лектора курса: «Базовые понятия и принципы ООП в C#» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).

1. Сайт Metanit.com
2. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
3. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
4. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
5. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
6. Корпорация Microsoft. C#. Спецификация языка.

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы
3. Программный код для каждого из заданий с подробными комментариями

Задание 1. Вычислить наибольший общий делитель двух целых чисел с помощью алгоритма Евклида

In this exercise, you will write a method that implements Euclid's algorithm for calculating the GCD of two integers passed in as parameters. You will test this method by using a Windows® Presentation Foundation (WPF) application that prompts the user for the parameter values, and displays the result. You will also generate a unit test project to enable you to automate testing this method.

Scenario

Some of the data that is collected by devices built by Fabrikam, Inc. must be encrypted for security purposes. Encryption algorithms often make use of prime numbers. A part of the algorithm that generates prime numbers needs to calculate the GCD of two numbers.

The GCD of two numbers is the largest number that can exactly divide into the two numbers. For example, the GCD of 15 and 12 is 3. Three is the largest whole number that divides exactly into 15 and 12.

The process for finding the GCD of 2806 and 345 by using Euclid's algorithm is as follows.

1. Keep taking 345 away from 2806 until less than 345 is left and store the remainder.In this case,

2806 = (8 × 345) + 46, so the remainder is 46.

2. Keep taking the remainder (46) away from 345 until less than 46 is left, and store the remainder.

345 = (7 × 46) + 23, so the remainder is 23.

3. Keep taking 23 away from 46 until less than 23 is left, and store the remainder.

46 = (2 × 23) + 0

4. The remainder is 0, so the GCD of 2806 and 345 was the value of the previously stored remainder, which was 23 in this case.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Implement Euclid's algorithm.
3. Test the **FindGCDEuclid** method.
4. Create a unit test for the **FindGCDEuclid** method.

- Task 1: Implement Euclid's algorithm

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 1, Task 1** task. This task is located in the GCDAlgorithms.cs file.
3. In the **GCDAlgorithms** class, remove the **TODO Exercise 1, Task 1** comment and declare a **public static** method called **FindGCDEuclid**. The method should accept two integer parameters called *a* and *b*, and return an integer value.
4. In the **FindGCDEuclid** method, add code that calculates and returns the GCD of the values specified by the parameters *a* and *b* by using Euclid's algorithm.

Euclid's algorithm works as follows:

a. If *a* is zero, the GCD of *a* and *b* is *b*.

b. Otherwise, repeatedly subtract *b* from *a* (when *a* is greater than *b*) or subtract *a* from *b* (when *b* is greater than *a*) until *b* is zero.

c. The GCD of the two original parameters is the new value in *a*.

Код, реализующий метод Евклида приведен ниже. В отчете к лабораторной работе необходимо привести пример реализации этого кода, выполненного вручную.

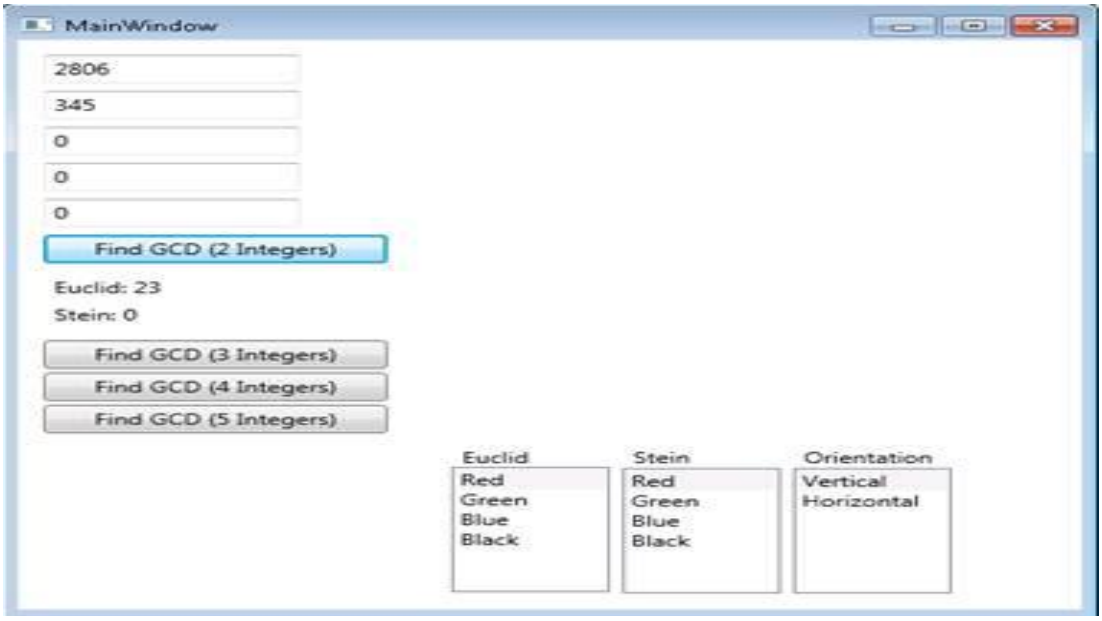
```
public static int NOD ( int a, int b )//метод Евклида
        {
            if (a == 0) return b;
            while (b != 0)
            {
                if (a > b)
                {
                    a -= b;
                }
                else
                {
                    b -= a;
                }
            }
            return a;
        }
```

- Task 2: Test the FindGCDEuclid method

1. Use the Task List window to navigate to the **TODO Exercise 1, Task 2** task. This task is located in the MainWindow.xaml.cs file. This is the code-behind file for a WPF window that you will use to test the **FindGCDEuclid** method and display the results.
2. Remove the **TODO Exercise 1, Task 2** comment, add code to call the static **FindGCDEuclid** method of the **GCDAlgorithms** class, and display the results in the **resultEuclid** label control. In the method call, use the firstNumber and secondNumber variables as arguments (these variables contain values that the user enters in the WPF window). Finally, the result should be formatted as the following code example shows.

Hint: Set the **Content** property of a label control to display data in a label. Use the **String.Format** method to create a formatted string.

3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first text box, type 2806
6. In the second text box, type 345 and then click Find GCD (2 Integers). The result of 23 should be displayed, as the following screen shot shows.



7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

First number	Second number	Result
0	0	0
0	10	10
25	10	5
25	100	25
26	100	2
27	100	1

8. Close the GreatestCommonDivisor application.

- Task 4: Create a unit test for the FindGCDEuclid method

1. Open the GCDAlgorithms.cs file.
2. In the GCDAlgorithms class, create a unit test for the FindGCDEuclid method. Create a new Test Project called GCD Test Project to hold the unit test.
3. In the GCD Test Project project, in the GCDAlgorithmsTest.cs file, locate the FindGCDEuclidTest method.
4. In the FindGCDEuclidTest method, set the a variable to 2806, set the b variable to 345, set the expected variable to 23, and then remove the Assert.Inconclusive method call.
5. Open the Test View window and refresh the display if the unit test is not listed.
6. Run the FindGCDEuclidTest test and verify that the test ran successfully.

Задание 2. Вычислить наибольший общий делитель 3,4 и 5 чисел с помощью перегрузки методов

In this exercise, you will create overloaded versions of this method that can take three, four, or five integer parameters and calculate the GCD of all of these parameters.

Scenario

Some of the encryption algorithms used by devices that Fabrikam, Inc. builds require calculating the GCD of sets of numbers, not just pairs. You have been asked to provide implementations of the Euclid algorithm that can calculate the GCD of three, four, or five integers.

The process for finding the GCD of three numbers *x*, *y*, and *z* is straightforward:

1. Calculate the GCD of *x* and *y* by using the algorithm for two numbers, and store the result in a variable r.
2. Calculate the GCD of r and *z*. The result is the GCD of *x*, *y*, and *z*.

You can apply the same technique to calculate the GCD of four or five integers:

- GCD(*w*, *x*, *y*, *z*) = GCD(*w*, GCD(*x*, *y*, *z*))
- GCD(*v*, *w*, *x*, *y*, *z*) = GCD(*v*, GCD(*w*, *x*, *y*, *z*))

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add overloaded methods to the **GCDAlgorithms** class.
3. Test the overloaded methods.
4. Create unit tests for the overloaded methods.

- Task 1: Open the starter project

- Task 2: Add overloaded methods to the GCDAlgorithms class

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 2** task.
3. In the **GCDAlgorithms** class, remove the **TODO Exercise 2, Task 2** comment, and then declare an overloaded version of the **FindGCDEuclid** method. The method should accept three integer parameters called a, b, and c, and return an integer value.
4. In the new method, add code that uses the original **FindGCDEuclid** method, to find the GCD for the parameters *a* and *b*. Store the result in a new variable called d.
5. Add a second call to the original **FindGCDEuclid** method to find the GCD for variable d and parameter *c*. Store the result in a new variable called e.
6. Add code to return the parameter *e* from the **FindGCDEuclid** method.
7. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept four integer parameters called *a*, *b*, *c*, and *d*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.
8. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept five integer parameters called *a*, *b*, *c*, *d*, and *e*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.

- Task 3: Test the overloaded methods

1. Use the Task List window to navigate to the **TODO Exercise 2, Task 3** task.

This task is located in the code for the WPF window that you can use to test your code.

2. Remove the **TODO Exercise 2, Task 3** comment, locate the **else if (sender == findGCD3)** block, and modify the statement that sets the **Content** property of the **resultEuclid** label to "N/A" as follows:

a. Call the **FindGCDEuclid** overload that accepts three parameters and pass the variables firstNumber, secondNumber, and thirdNumber as arguments.

b. Display the results in the **resultEuclid** label control. The result should be formatted as the following code example shows.

```
Euclid: result
```

3. Locate the **else if (sender == findGCD3)** block, the **else if (sender == findGCD4)** block, and the **else if (sender == findGCD5)** block, and modify the statements that set the **Content** property of the **resultEuclid** label to "N/A". Call the appropriate **FindGCDEuclid** overload by using the firstNumber, secondNumber, thirdNumber, fourthNumber, and fifthNumber variables as arguments. Display the results in the **resultEuclid** label control.

4. Build the solution and correct any errors.

5. Run the GreatestCommonDivisor application.

6. In the GreatestCommonDivisor application, in the MainWindow window, type the values **7396 1978 1204 430 258** and then click **Find GCD (5 Integers)**.

Verify that the result **86** is displayed.

7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

First number	Second number	Third number	Fourth number	Fifth number	Result
2806	345	0	0	0	23
0	0	0	0	0	0
0	0	0	0	1	1
12	24	36	48	60	12
13	24	36	48	60	1
14	24	36	48	60	2
15	24	36	48	60	3
16	24	36	48	60	4
0	24	36	48	60	12

- Task 4: Create unit tests for the overloaded methods

1. In Visual Studio, review the task list.

2. Use the Task List window to navigate to the **TODO Exercise 2, Task 4** task.

3. Remove the **TODO Exercise 2, Task 4** comment and add a test method called **FindGCDEuclidTest1**.

4. In the **FindGCDEuclidTest1** method, declare four variables called a, b, c, and expected, and assign them values **7396**, **1978**, **1204**, and **86** respectively.

5. Declare a variable called actual, and assign it the result of a call to the **FindGCDEuclid** method call. Use the variables a, b, and c as arguments.

6. Call the **AreEqual static** method of the **Assert** class, and pass the expected and actual variables as arguments.

7. Repeat steps 4–6 to create two more test methods to test the other **FindGCDEuclid** method overloads. Create test methods called **FindGCDEuclidTest2** and **FindGCDEuclidTest3**. Use the values **7396**, **1978**, **1204**, and **430** for the **FindGCDEuclidTest2** method, and the values **7396**, **1978**, **1204**, **430**, and **258** for the **FindGCDEuclidTest3** method. The result should be **86** in both cases.

8. Open the Test View window and refresh the display if the unit test is not listed.

9. Run the FindGCDEuclidTest, FindGCDEuclidTest1, FindGCDEuclidTest2, and FindGCDEuclidTest3 tests and verify that the tests ran successfully.

Обязательное дополнение. Необходимо видоизменить логику программы и предусмотреть возможность нахождения наименьшего общего делителя у большого числа чисел, записанных в текстовом виде с использованием разделителя текста. Предлагаем использовать Params для передачи в метод неопределенного числа параметров. Используя Split и выбранный вами разделитель, например пробел или запятую, разделить строку на массив строк. Затем преобразовать этот массив в массив целых чисел и передать с помощью Params в метод нахождения общего делителя. Для сокращения кода метода вычисления НОД используйте рекурсию. Попробуйте перегрузить Params, это возможно?

Приветствуется использование кортежей для возврата из метода нескольких значений, например возврат НОД и времени выполнения функции.

Задание 3. Сравнить эффективность двух алгоритмов (Евклида и Штейна)

Алгоритм Штейна основан на использовании операций побитового сдвига. Напомним, сдвиг влево эквивалентен умножению на 2 (c<<1 равно c*2).

Обязательное тренировочное задание. Написать программу нахождения самого большого простого числа меньшего заданного целого числа N, и представить его в помощью цифры 2 и побитового сдвига.

In this exercise, you will write another method that implements Stein's algorithm for calculating the GCD of two integer parameters. The method will take an output parameter that contains the time taken to perform the calculation. You will also modify the method that implements Euclid's algorithm for calculating the GCD of two parameters to take an output parameter, also containing the time taken to perform the calculation. You will then modify the WPF application to test the relative performance of the methods and display the times taken.

Scenario

Stein's algorithm is an alternative algorithm for finding the GCD of two numbers. You have been told that it is more efficient than Euclid's algorithm. A colleague has previously implemented Stein's algorithm, but you decide to test this hypothesis by comparing the time taken to calculate the GCD of pairs of numbers with that taken by using Euclid's algorithm.

The following steps describe the process of calculating the GCD of two numbers, u and v, by following Stein's algorithm:

1. $\gcd(0, v) = v$ because everything divides by zero, and v is the largest number that divides v.

Similarly, $\gcd(u, 0) = u$. $\gcd(0, 0)$ is not typically defined, but it is convenient to set $\gcd(0, 0) = 0$.

2. If u and v are both even, $\gcd(u, v) = 2 \cdot \gcd(u/2, v/2)$ because 2 is a common divisor.

3. If u is even and v is odd, $\gcd(u, v) = \gcd(u/2, v)$ because 2 is not a common divisor.

Similarly, if u is odd and v is even, $\gcd(u, v) = \gcd(u, v/2)$.

4. If u and v are both odd, and $u \geq v$, $\gcd(u, v) = \gcd((u - v)/2, v)$.

If both are odd and $u < v$, $\gcd(u, v) = \gcd((v - u)/2, u)$.

These are combinations of one step of the simple Euclidean algorithm, which uses subtraction at each step, and an application of step 4 above. The division by 2 results in an integer because the difference of two odd numbers is even.

5. Repeat steps 3–5 until $u = v$, or (one more step) until $u = 0$.

In either case, the result is $2kv$, where k is the number of common factors of 2 found in step 2.

The main tasks for this exercise are as follows:

1. Open the starter project.

2. Implement Stein's algorithm.
3. Test the **FindGCDStein** method.
4. Add code to test the performance of the algorithms.

- Task 1: Open the starter project
- Open the Stein solution in the E:\Labfiles\Lab 3\Ex3\Starter folder.

This solution contains a completed copy of the code from Exercise 2.

- Task 2: Implement Stein's algorithm

1. Open the GCDAlgorithms.cs file.
2. At the end of the **GCDAlgorithms** class, remove the TODO comment and declare a **public static** method called **FindGCDStein**. The method should accept two integer parameters called *u* and *v*, and return an integer value.
3. In the **FindGCDStein** method, add the code in the following code example, which calculates and returns the GCD of the values that are specified by the parameters *u* and *v* by using Stein's algorithm. You can either type this code manually, or use the Mod03Stein code snippet.

Note: For the purposes of this exercise, it is not necessary for you to understand this code. However, if you have time, you may like to compare this method to the algorithm that is described in the exercise scenario. Note that this code uses the left-shift (<<) and right-shift (>>) operators to perform fast multiplication and division by 2. If you left-shift an integer value by one place, the result is the same as multiplying the integer value by 2. Similarly, if you right-shift an integer value by one place, the result is the same as dividing the integer value by 2. In addition, the **|** operator performs a bitwise **OR** operation between two integer values. Consequently, if either *u* or *v* are zero, the expression **u | v** is a fast way of returning the value of whichever variable is non-zero, or zero if both are zero. Similarly, the **&** operator performs a bitwise **AND** operation, so the expression **u & 1** is a fast way to determine whether the value of *u* is odd or even.

```
static public int FindGCDStein(int u, int v)
{
    int k;
    // Step 1.
    // gcd(0, v) = v, because everything divides zero,
    // and v is the largest number that divides v.
    // Similarly, gcd(u, 0) = u. gcd(0, 0) is not typically
    // defined, but it is convenient to set gcd(0, 0) = 0.
    if (u == 0 || v == 0)
        return u | v;
    // Step 2.
    // if u and v are both even, then gcd(u, v) = 2•gcd(u/2, v/2),
    // because 2 is a common divisor.
    for (k = 0; ((u | v) & 1) == 0; ++k)
    {
        u >>= 1;
        v >>= 1;
    }
    // Step 3.
    // if u is even and v is odd, then gcd(u, v) = gcd(u/2, v),
    // because 2 is not a common divisor.
    // Similarly, if u is odd and v is even,
    // then gcd(u, v) = gcd(u, v/2).

    while ((u & 1) == 0)
        u >>= 1;
    // Step 4.
    // if u and v are both odd, and u ≥ v,
    // then gcd(u, v) = gcd((u – v)/2, v).
    // If both are odd and u < v, then gcd(u, v) = gcd((v – u)/2, u).
    // These are combinations of one step of the simple
    // Euclidean algorithm,
```

```
            // which uses subtraction at each step, and an application
            // of step 3 above.
            // The division by 2 results in an integer because the
            // difference of two odd numbers is even.
            do
            {
                while ((v & 1) == 0)  // Loop x
                    v >>= 1;
                // Now u and v are both odd, so diff(u, v) is even.
                //   Let u = min(u, v), v = diff(u, v)/2.
                if (u < v)
                {
                    v -= u;
                }
                else
                {
                    int diff = u - v;
                    u = v;
                    v = diff;
                }
                v >>= 1;
                // Step 5.
                // Repeat steps 3–4 until u = v, or (one more step)
                // until u = 0.
                // In either case, the result is (2^k) * v, where k is
                // the number of common factors of 2 found in step 2.
            } while (v != 0);
            u <<= k;
            return u;
        }
```

- Test the FindGCDStein method

1. Open the MainWindow.xaml.cs file.
2. In the **MainWindow** class, in the **FindGCD_Click** method, locate the **TODO Exercise 3, Task 2** comment. Remove this comment and replace the statement that sets the **Content** property of the **resultStein** label with code that calls the **FindGCDStein** method by using the variables firstNumber and secondNumber as arguments. Display the results in the **resultStein** label control. The result should be formatted as the following code example shows.

Stein: *result*

3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**.

Verify that the value **4** is displayed in both labels.

6. Close the GreatestCommonDivisor application.
7. Open the GCDAlgorithmsTest.cs file.
8. At the end of the **GCDAlgorithmsTest** class, locate the **TODO Exercise 3, Task 2** comment, remove the comment, and then add a test method called **FindGCDSteinTest**.
9. In the **FindGCDSteinTest** method, declare three variables called u, v, and expected, and assign them values **298467352**, **569484**, and **4** respectively.
10. Declare a variable called actual, and assign it the result of a call to the **FindGCDStein** method call. Use the variables u and v as arguments.
11. Call the **static AreEqual** method of the **Assert** class, and pass the expected and actual variables as arguments.
12. Open the Test View window and refresh the display if the unit test is not listed.
13. Run the FindGCDSteinTest test, and verify that the test ran successfully.

- Task 4: Add code to test the performance of the algorithms

1. Open the GCDAlgorithms.cs file.

2. In the **GCDAlgorithms** class, locate the **FindGCDEuclid** method that accepts two parameters, and modify the method signature to take an *out* parameter called *time* of type **long**.
3. At the start of the **FindGCDEuclid** method, add code to initialize the time parameter to zero, create a new **Stopwatch** object called **sw**, and start the stop watch.

The **Stopwatch** class is useful for timing code. The **Start** method starts an internal timer running. You can subsequently use the **Stop** method to halt the timer, and establish how long the interval was between starting and stopping the timer by querying the **ElapsedMilliseconds** or **ElapsedTicks** properties.

4. At the end of the **FindGCDEuclid** method, before the **return** statement, add code to stop the **Stopwatch** object, and set the *time* parameter to the number of elapsed ticks of the **Stopwatch** object.
5. Comment out the other **FindGCDEuclid** method overloads.
6. Modify the **FindGCDStein** method to include the time output parameter, and add code to record the time each method takes to run. Note that the **FindGCDStein** method contains two **return** statements, and you should record the time before each one.
7. Open the MainWindow.xaml.cs file.
8. In the **FindGCD**_Click method, modify each of the calls to the **FindGCDEuclid** method and the **FindGCDStein** method to use the updated method signatures, as follows:
   a. For calling the Euclid algorithm, create a long variable called timeEuclid.
   b. For calling the Stein algorithm, create a long variable called timeStein.
   c. Format the results displayed in the labels as the following code example shows.

```
[Euclid]
Euclid: result, Time (ticks): result

[Stein]
Stein: result, Time (ticks): result
```

9. Comment out the code that calls the overloaded versions of the **FindGCDEuclid** method.
10. Open the GCDAlgorithmsTest.cs file.
11. Modify the **FindGCDEuclidTest** and **FindGCDSteinTest** methods to use the new method signatures. Comment out the methods **FindGCDEuclidTest1**, **FindGCDEuclidTest2**, and **FindGCDEuclidTest3**.
12. Build the solution and correct any errors.
13. Run the GreatestCommonDivisor application.
14. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The result of **4** should be displayed. The time reported for Euclid's algorithm should be approximately three times more than that for Stein's algorithm.

Note: The bigger the difference between the two values, the more efficient Stein's algorithm becomes compared to Euclid's. If you have time, try experimenting with different values.

15. Close the GreatestCommonDivisor application.
16. Open the Test View window and refresh the display if the unit test is not listed.
17. Run the **FindGCDEuclidTest** and **FindGCDSteinTest** methods and verify that the tests ran successfully.

Задание 4: Умножение матриц

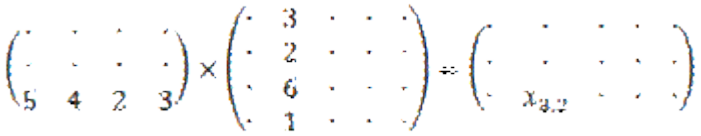
In this exercise, you will create another WPF application. This WPF application will provide a user interface that enables the user to provide the data for two matrices and store this data in rectangular arrays. The application will calculate the product of these two arrays and display them.

Scenario

Some of the devices that Fabrikam, Inc. has developed perform calculations that involve sets of data that are held as matrices. You have been asked to implement code that performs matrix multiplication. You decide to test your code by building a WPF application

that enables a user to specify the data for two matrices, calculate the product of these matrices, and then view the result.

Multiplying matrices is an iterative process that involves calculating the sum of the products of the values in each row in one matrix with the values in each column in the other, as the following screen shot shows.



This screen shot shows a 3×4 matrix multiplying a 4×5 matrix. This will result in a 3×5 matrix.

Note: The number of columns in the first matrix must match the number of rows in the second matrix. The starter code that is provided for you in this lab ensures that this is always the case.

To calculate each element xa,b in the result matrix, you must calculate the sum of the products of every value in row a in the first matrix with every value in column b in the second matrix. For example, to calculate the value placed at x3,2 in the result matrix, you calculate the sum of the products of every value in row 3 in the first matrix with every value in column 2 in the second matrix:

(5×3)+(4×2)+(2×6)+(3×1) = 38

You perform this calculation for every element in the result matrix.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication project and examine the starter code.
2. Define the matrix arrays and populate them with the data in the **Grid** controls.
3. Multiply the two input matrices and calculate the result.
4. Display the results and test the application.

- Task 1: Open the MatrixMultiplication project and examine the starter code

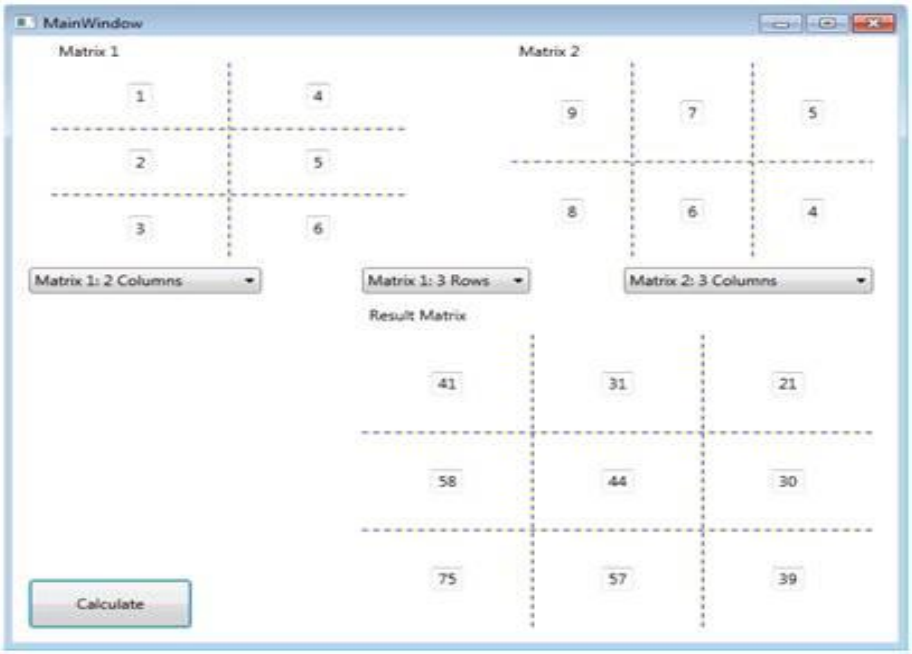
Examine the user interface that the MainWindow window defines.

The user interface contains three **Grid** controls, three **ComboBox** controls, and a **Button** control.

When the application runs, the first **Grid** control, labeled **Matrix 1**, represents the first matrix, and the second **Grid** control, labeled **Matrix 2**, represents the second matrix. The user can specify the dimensions of the matrices by using the **ComboBox** controls, and then enter data into each cell in them. There are several rules that govern the compatibility of matrices to be multiplied together, and Matrix 2 is automatically configured to have an appropriate number of rows based on the number of columns in Matrix 1.

When the user clicks the **Calculate** button, Matrix 1 and Matrix 2 are multiplied together, and the result is displayed in the **Grid** control labeled **Result Matrix**. The dimensions of the result are determined by the shapes of Matrix 1 and Matrix 2.

The following screen shot shows the completed application running. The user has multiplied a 2×3 matrix with a 3×2 matrix, and the result is a 3×3 matrix.



- Task 2: Define the matrix arrays and populate them with the data in the Grid controls

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file.
3. At the top of the **MainWindow** class, remove the comment **TODO Task 2 declare variables**, and then add statements that declare three two-dimensional arrays called **matrix1**, **matrix2**, and **result**. The type of the elements in these arrays should be **double**, but the size of each dimension should be omitted because the arrays will be dynamically sized based on the input that the user provides. The first dimension will be set to the number of columns, and the second dimension will be set to the number of rows.
4. In the task list, double-click the task **TODO Task 2 Copy data from input Grids**. This task is located in the **buttonCalculate_Click** method.
5. In the **buttonCalculate_Click** method, remove the comment **TODO Task 2 Copy data from input Grids**. Add two statements that call the **getValuesFromGrid** method. This method (provided in the starter code) expects the name of a **Grid** control and the name of an array to populate with data from that **Grid** control. In the first statement, specify that the method should use the data in **grid1** to populate **matrix1**. In the second statement, specify that the method should use the data from **grid2** to populate **matrix2**.
6. Remove the comment **TODO Task 2 Get the matrix dimensions**. Declare three integer variables called m1columns_m2rows, m1rows, and m2columns. Initialize m1columns_m2rows with the number of columns in the **matrix1** array (this is also the same as the number of rows in the **matrix2** array) by using the **GetLength** method of the first dimension of the array. Initialize m1rows with the number of rows in the **matrix1** array by using the **GetLength** method of the second dimension of the array. Initialize m2columns with the number of columns in the **matrix2** array.

- Task 3: Multiply the two input matrices and calculate the result

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 3 Calculate the result**. Define a **for** loop that iterates through all of the rows in the **matrix1** array. The dimensions of an array are integers, so use an integer variable called row as the control variable in this **for** loop. Leave the body of the **for** loop blank; you will add code to this loop in the next step.
2. In the body of the **for** loop, add a nested **for** loop that iterates through all of the columns in the **matrix2** array. Use an integer variable called column as the control variable in this **for** loop. Leave the body of this **for** loop blank.
3. The contents of each cell in the **result** array are calculated by adding the product of each item in the row identified by the row variable in **matrix1** with each item in the column identified by the column variable in **matrix2**. You will require another loop to perform this calculation, and a variable to store the result as this loop calculates it.

In the inner **for** loop, declare a double variable called accumulator, and then initialize it to zero.

4. Add another nested **for** loop after the declaration of the accumulator variable. This loop should iterate through all of the columns in the current row in the **matrix1** array. Use an integer variable called cell as the control variable in this **for** loop. Leave the body of this **for** loop blank.
5. In the body of this **for** loop, multiply the value in **matrix1[cell, row]** with the value in **matrix2[column, cell]**, and then add the result to accumulator.
6. After the closing brace of the innermost **for** loop, store the value in accumulator in the **result** array. The value should be stored in the cell that the column and row variables have identified.

- Task 4: Display the results and test the application

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 4 Display the result**. The starter code contains a method called **initializeGrid** that displays the contents of an array in a **Grid** control in the WPF window. Add a statement that calls this method. Specify that the method should use the **grid3 Grid** control to display the contents of the **result** array.
2. Build the solution and correct any errors.
3. Run the application in Debug mode.
4. In the MainWindow window, define **Matrix 1** as a 3×2 matrix and define **Matrix 2** as a 3×3 matrix.

Note: The number of rows in the **Matrix 2** matrix is determined by the number of columns in the **Matrix 1** matrix.

Замечание 1: при инициализации умножаемых матриц заполнить ячейки случайными целыми или вещественными числами.

Замечание 2: предусмотреть полосы прокрутки для компонентов, отображающих матрицы в случае большого числа строк или столбцов.

Примечание: Для графического отображения матрицы можно использовать компонент Grid, или компонент DataGridView. Если требуемого компонента нет на панели элементов (он может отсутствовать на панели элементов в зависимости от типа проекта, который вы создаете WPF или Windows Forms) , воспользоваться им можно программно, создавая его динамически. Но проще добавить компонент на панель элементов, для этого щелкните правой кнопкой мыши в любом месте панели элементов и выберите вкладку «Выбрать элементы», затем вкладку «Компоненты .Net Framework». Ниже приведен метод инициализации Grid, его очистка и заполнение строками и столбцами:

```
private void initializeGrid(Grid grid, double[,] matrix)
    {       if (grid != null)
        {
            // Reset the grid before doing anything
            grid.Children.Clear();
            grid.ColumnDefinitions.Clear();
            grid.RowDefinitions.Clear();
            // Get the dimensions
            int columns = matrix.GetLength(0);
            int rows = matrix.GetLength(1);
            // Add the correct number of coumns to the grid
            for (int x = 0; x < columns; x++)
            {
```

```
                        grid.ColumnDefinitions.Add(new ColumnDefinition() { Width
= new GridLength(1, GridUnitType.Star), });
                    }
                    for (int y = 0; y < rows; y++)
                    {
// GridUnitType.Star - The value is expressed as a weighted proportion of available space
                        grid.RowDefinitions.Add(new RowDefinition() { Height = new
GridLength(1, GridUnitType.Star), });
                    }
// Fill each cell of the grid with an editable TextBox containing the value from the matrix
                    for (int x = 0; x < columns; x++)
                    {       for (int y = 0; y < rows; y++)
                        {
                            double cell = (double)matrix[x, y];
                            TextBox t = new TextBox();
                            t.Text = cell.ToString();
          t.VerticalAlignment = System.Windows.VerticalAlignment.Center;
          t.HorizontalAlignment = System.Windows.HorizontalAlignment.Center;
                            t.SetValue(Grid.RowProperty, y);
                            t.SetValue(Grid.ColumnProperty, x);
                            grid.Children.Add(t);        }           }         }
}
```

Ниже приведен текст программы, реализующей функцию считывания значений из текстовых полей сетки в двумерный массив вещественных значений:

```
      private void getValuesFromGrid(Grid grid, double[,] matrix)         {
                  int columns = grid.ColumnDefinitions.Count;
                  int rows = grid.RowDefinitions.Count;
                 // Iterate over cells in Grid, copying to matrix array
                  for (int c = 0; c < grid.Children.Count; c++)
                  {   TextBox t = (TextBox)grid.Children[c];
                      int row = Grid.GetRow(t);
                      int column = Grid.GetColumn(t);
                      matrix[column, row] = double.Parse(t.Text);   }        }
```

На наш взгляд, проще работать с компонентом DataGridView:
using System.Data; // - для DataGridView
Создаем экземпляр класса DataTable Таблица = new DataTable();
Заполняем таблицу Таблица.Rows[j][i] = A[j,i];
Визуализируем таблицу: dataGridView1.DataSource = Таблица;
Этот пример под номером 59 есть в книге Зиборов В. В.Visual C# 2012 на примерах. — СПб.: БХВ-Петербург, 2013. — 480 с.: ил.

Практическая работа №3

Тема: Обработка исключительных ситуаций

Цель работы: Научиться на практике обрабатывать исключения, генерировать собственные исключения, отлавливать исключения различных типов, проверять числовые данные на выход за границы значений.

Ключевые понятия: Exception, Try/Catch, Finally, Checked и Unchecked, throw.

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентацию лектора курса: «Обработка исключений в C# » (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка.

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы

Тренировочное задание – обязательное!

Используя приведенный ниже готовый код, создайте консольное приложение и осуществите несколько экспериментов.

1. Запустите программу, введите значение i=0. Убедитесь, что программа работает, но во внешнем блоке не выводятся параметры исключения. Почему?
2. Закомментируйте блок catch (System.DivideByZeroException e).

Убедитесь, что параметры исключения выводятся на печать. Опишите все приведенные параметры исключения.

3. Поставьте на первое место блок catch с общим исключением. Что произойдет?
4. Можно убрать все внутренние блоки catch? Можно ли оставить только блок Try?
5. Если исключение в блоке Try – Catch не поймано, означает ли это крах программы. Можно ли написать программу, которая никогда аварийно не завершается?

```
try
            {
                try
                {
                    x = 5;
                    y = x / i;
                    Console.WriteLine("x={0}, y= {1}", x, y);
                }
                catch (System.DivideByZeroException e)
                {
                    Console.WriteLine("Попытка деления на ноль", e.ToString());
                }
                catch (System.FormatException e)
                {
               Console.WriteLine("Введено не целое число! Исключение", e.ToString());
                }
                catch
                {
                    Console.WriteLine("Неизвестная ошибка. Перезапустите программу");
                    throw;
                }
                finally
                {
                    Console.WriteLine("Выполнили блок finally");
```

```
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine(ex.StackTrace);
        Console.WriteLine(ex.TargetSite);
        Console.WriteLine(ex.InnerException);
        Console.WriteLine(ex.Source);
        Console.WriteLine(ex.Data);
        Console.WriteLine(ex.HelpLink);
          }
```

Задание 1. Написать с использованием конструкций Switch, Try, Catch метод анализа опасных состояний оборудования компьютера

In this exercise, you will add fail-safe functionality to an application to ensure that it continues to function even if one or more exceptions occur. The code itself is located in a Windows Presentation Foundation (WPF) application that acts as a test harness.

Scenario

Fabrikam, Inc. provides intelligent switching devices that can monitor the environment for a critical condition (such as the temperature exceeding a specified value), and trigger a shutdown operation. These switching devices are used in applications in the energy industry to initiate the shutdown of nuclear reactors. Needless to say, the correct operation of these devices is essential. Fabrikam, Inc. is developing a new model of switching device, and requires you to write part of the software that controls its operation. You have been provided with the code that performs the shutdown operation. This code contains a number of steps, and they must all be run. If any step fails, the code must report the failure, but continue with the next step.

The main tasks for this exercise are as follows:

1. Open the Failsafe solution and run the application.
2. Examine the **Switch** class.
3. Handle the exceptions that the **Switch** class throws.
4. Test the application.

- Task 1: Open the Failsafe solution and run the application

Note: The **Switch** class is designed to randomly throw an exception, so you may not encounter an exception the first time that you click the button. Repeatedly click the **Shutdown** button until an exception occurs.

- Task 2: Examine the Switch class

1. If it is not already open, open the Switch.cs file in Visual Studio.
2. Examine the **Switch** class.

Note that the class contains several methods, each of which is capable of throwing at least one exception, dependent on the outcome of a random number generation. Toward the bottom of the file, note the definitions of each of the custom exceptions that the **Switch** class can throw. These are very basic exception classes that simply encapsulate an error message.

- Task 3: Handle the exceptions that the Switch class throws

The SwitchTestHarness project contains a reference to the **SwitchDevice** class, and invokes each method in the **Switch** class to simulate polling multiple sensors and diagnostic devices. Currently, the project contains no exception handling, so when an exception occurs, the application will fail. You must add exception-handling code to the SwitchTestHarness project, to protect the application from exceptions that the **Switch** class throws.

1. Open the MainWindow.xaml.cs file in Visual Studio.
2. In the **MainWindow** class, locate the **Button1_Click** method. This method runs when the user clicks the **Shutdown** button.

3. Remove the comment **TODO - Add exception handling**, and then locate the **Step 1 - disconnect from the Power Generator** and **Step 2 - Verify the status of the Primary Coolant System** comments. Enclose the code between these comments in a **try/catch** block that catches the **SwitchDevices.PowerGeneratorCommsException** exception. This is the exception that the **DisconnectPowerGenerator** method can throw.
4. In the **catch** block, add code to append a new line of text to the **textBlock1** control with the message "*** Exception in step 1:" and then the contents of the **Message** property of the exception. The **Message** property contains the error message that the **Switch** object specified when it threw the exception.

Hint: To append a line of text to a **TextBlock** control, use the **+=** operator on the **Text** property of the control.

5. Enclose the code between the **Step 2 - Verify the status of the Primary Coolant System** and **Step 3 - Verify the status of the Backup Coolant System** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, following the same pattern as step 3, print a message on a new line in the **textBlock1** control (note that this is step 2, not step 1 of the shutdown process).
6. Enclose the code between the **Step 3 - Verify the status of the Backup Coolant System** and **Step 4 - Record the core temperature prior to shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 3).
7. Enclose the code between the **Step 4 - Record the core temperature prior to shutting down the reactor** and **Step 5 - Insert the control rods into the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 4).
8. Enclose the code between the **Step 5 - Insert the control rods into the reactor** and **Step 6 - Record the core temperature after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.RodClusterReleaseException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 5).
9. Enclose the code between the **Step 6 - Record the core temperature after shutting down the reactor** and **Step 7 - Record the core radiation levels after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 6).
10. Enclose the code between the **Step 7 - Record the core radiation levels after shutting down the reactor** and **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.CoreRadiationLevelReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 7).
11. Enclose the two statements after **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.SignallingException** exception. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 8).
12. Build the solution and correct any errors.

- Task 4: Test the application
- Run the application, and then click the **Shutdown** button. Examine the messages displayed in the MainWindow window, and verify that exceptions are now caught and reported.

Note: The **Switch** class randomly generates exceptions as before, so you may not see any exception messages the first time that you click the button. Repeat the process of clicking the button and examining the output until you see exception messages appear.

Ниже приведен текст программы, реализации класса Switch и генератора исключений:

```
public enum CoolantSystemStatus { OK, Check, Fail }
public enum SuccessFailureResult { Success, Fail }
public class Switch
 {
     /// <summary>
     /// Utilty object for simulation
     /// </summary>
     private Random rand = new Random();
     /// <summary>
     /// Disconnect from the external power generation systems
     /// </summary>
     /// <returns>Success or Failure</returns>
     /// <exception cref="PowerGeneratorCommsException">Thrown when the
physical switch cannot establish a connection to the power generation
system</exception>
     public SuccessFailureResult DisconnectPowerGenerator()
     {
         SuccessFailureResult r = SuccessFailureResult.Fail;
         if (rand.Next(1, 10) > 2) r = SuccessFailureResult.Success;
         if (rand.Next(1, 20) > 18) throw new
PowerGeneratorCommsException("Network failure accessing Power Generator monitoring
system");
         return r;
     }
     /// <summary>
     /// Runs a diagnostic check against the primary coolant system
     /// </summary>
     /// <returns>Coolant System Status (OK, Fail, Check)</returns>
     /// <exception cref="CoolantTemperatureReadException">Thrown when the
switch cannot read the temperature from the coolant system</exception>
     /// <exception cref="CoolantPressureReadException">Thrown when the
switch cannot read the pressure from the coolant system</exception>
     public CoolantSystemStatus VerifyPrimaryCoolantSystem()
     {
         CoolantSystemStatus c = CoolantSystemStatus.Fail;
         int r = rand.Next(1, 10);
         if (r > 5)
         {
             c = CoolantSystemStatus.OK;
         }
         else if (r > 2)
         {
             c = CoolantSystemStatus.Check;
         }
         if (rand.Next(1, 20) > 18) throw new
CoolantTemperatureReadException("Failed to read primary coolant system temperature");
         if (rand.Next(1, 20) > 18) throw new
CoolantPressureReadException("Failed to read primary coolant system pressure");
         return c;
     }
     /// <summary>
     /// Runs a diagnostic check against the backup coolant system
     /// </summary>
     /// <returns>Coolant System Status (OK, Fail, Check)</returns>
     /// <exception cref="CoolantTemperatureReadException">Thrown when the
switch cannot read the temperature from the coolant system</exception>
     /// <exception cref="CoolantPressureReadException">Thrown when the
switch cannot read the pressure from the coolant system</exception>
     public CoolantSystemStatus VerifyBackupCoolantSystem()
     {
         CoolantSystemStatus c = CoolantSystemStatus.Fail;
```

```
                int r = rand.Next(1, 10);
                if (r > 5)
                {
                    c = CoolantSystemStatus.OK;
                }
                else if (r > 2)
                {
                    c = CoolantSystemStatus.Check;
                }
                if (rand.Next(1, 20) > 19) throw new
CoolantTemperatureReadException("Failed to read backup coolant system temperature");
                if (rand.Next(1, 20) > 19) throw new
CoolantPressureReadException("Failed to read backup coolant system pressure");
                return c;
            }

            /// <summary>
            /// Reads the temperature from the reactor core
            /// </summary>
            /// <returns>Temperature</returns>
            /// <exception cref="CoreTemperatureReadException">Thrown when the
switch cannot access the temperature data</exception>
            public double GetCoreTemperature()
            {
                if (rand.Next(1, 20) > 18) throw new
CoreTemperatureReadException("Failed to read core reactor system temperature");
                return rand.NextDouble() * 1000;
            }

            /// <summary>
            /// Instructs the reactor to insert the control rods to shut the
reactor down
            /// </summary>
            /// <returns>Success or failure</returns>
            /// <exception cref="RodClusterReleaseException">Thrown if the switch
device cannot read the rod position</exception>
            public SuccessFailureResult InsertRodCluster()
            {
                SuccessFailureResult r = SuccessFailureResult.Fail;
                if (rand.Next(1, 100) > 5) r = SuccessFailureResult.Success;
                if (rand.Next(1, 10) > 8) throw new
RodClusterReleaseException("Sensor failure, cannot verify rod release");
                return r;
            }

            /// <summary>
            /// Reads the radiation level from the reactor core
            /// </summary>
            /// <returns>Temperature</returns>
            /// <exception cref="CoreRadiationLevelReadException">Thrown when the
switch cannot access the radiation level data</exception>
            public double GetRadiationLevel()
            {
                if (rand.Next(1, 20) > 18) throw new
CoreRadiationLevelReadException("Failed to read core reactor system radiation
levels");
                return rand.NextDouble() * 500;
            }

            /// <summary>
            /// Sends a broadcast message to PA system notofying shutdown complete
            /// </summary>
```

```
            /// <exception cref="SignallingException">Thrown if the switch cannot
connect to the PA system over the network</exception>
            public void SignalShutdownComplete()
            {
                if (rand.Next(1, 20) > 18) throw new SignallingException("Network
failure connecting to broadcast systems");
            }
        }
    public class PowerGeneratorCommsException : Exception
        {
            public PowerGeneratorCommsException(string message) : base(message) { }
        }
        public class CoolantSystemException : Exception
        {
            public CoolantSystemException(string message) : base(message) { }
        }
        public class CoolantTemperatureReadException : CoolantSystemException
        {
            public CoolantTemperatureReadException(string message) : base(message) { }
        }
        public class CoolantPressureReadException : CoolantSystemException
        {
            public CoolantPressureReadException(string message) : base(message) { }
        }
        public class CoreTemperatureReadException : Exception
        {
            public CoreTemperatureReadException(string message) : base(message) { }
        }
        public class CoreRadiationLevelReadException : Exception
        {
            public CoreRadiationLevelReadException(string message) : base(message) { }
        }
        public class RodClusterReleaseException : Exception
        {
            public RodClusterReleaseException(string message) : base(message) { }
        }
        public class SignallingException : Exception
        {
            public SignallingException(string message) : base(message) { }
        }
```

Задание 2. Обработка исключительных состояний при вычислении произведения матриц

In this exercise, you will modify a method so that it throws an **ArgumentException** exception if it is invoked with arguments that contain erroneous or invalid data.

Scenario

One of the engineering devices that Fabrikam, Inc. produces performs several calculations that involve matrices. These matrices represent the coordinates of sets of points within the bounds of a multidimensional mesh. The device itself collects the data for these points and constructs the matrices. Then, it uses a C# method to multiply them together to generate a new set of data points. Under normal operations, none of the data items in any of the matrices should be negative. However, sometimes the data that the device captures contains an error—if the device detects a value that is out of range, it generates the value –1 for a data point. Unfortunately, the code that multiplies matrices together fails to detect this condition, and calculates a result that is erroneous. You have been provided with a copy of this code as a method that is embedded in a WPF application.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication solution.
2. Add code to throw exceptions in the **MatrixMultiply** method.
3. Handle the exceptions that the **MatrixMultiply** method throws.
4. Implement test cases and test the application.

- Task 1: Open the MatrixMultiplication solution (lab2, task3)

The **MatrixMultiply** method performs the arithmetic to multiply together the two matrices passed as parameters and return the result.

Currently, the method accepts matrices of any size, and performs no validation of data in the matrices before calculating the results. You will add checks to ensure that the two matrices are compatible (the number of columns in the first matrix is equal to the number of rows in the second matrix), and that no value in either matrix is a negative number.

If the matrices are not compatible, or either of them contain a negative value, the method must throw an exception.

- Task 2: Add code to throw exceptions in the MatrixMultiply method

1. In the **MatrixMultiply** method, locate and remove the comment **TODO – Evaluate input matrices for compatibility**. Below the comment block, add code to perform the following actions:
   a. Compare the number of columns in **matrix1** to the number of rows in **matrix2**.
   b. Throw an **ArgumentException** exception if the values are not equal. The exception message should specify that the number of columns and rows should match.

Hint: You can obtain the number of columns in a matrix by examining the length of the first dimension. You can obtain the number of rows in a matrix by examining the length of the second dimension.

2. Locate and remove the comment **TODO – Evaluate matrix data points for invalid data**. At this point, the method iterates through the data points in each matrix, multiplying the value in each cell in **matrix1** against the value in the corresponding cell in **matrix2**. Add code below the comment block to perform the following actions:
   a. Check that the value in the current column and row of **matrix1** is greater than zero. The cell and row variables contain the column and row that you should examine.
   b. Throw an **ArgumentException** exception if the value is not greater than zero. The exception should contain the message "Matrix1 contains an invalid entry in cell[*x*, *y*]." where *x* and *y* are the column and row values of the cell.

Hint: Use the **String.Format** method to construct the exception message.

3. Add another block of code to check that the value in the current column and row of **matrix2** is greater than zero. If it is not, throw an **ArgumentException** exception with the message "Matrix2 contains an invalid entry in cell[*x*, *y*].". The column and cell variables contain the column and row that you should examine.

- Task 3: Handle the exceptions that the MatrixMultiply method throws

1. Open the MainWindow WPF window in the Design View window and examine the window.
   This window provides the user interface that enables the user to enter the data for the two matrices to be multiplied. The user clicks the **Calculate** button to calculate and display the result.
2. Open the code file for the MainWindow WPF window.
3. In the **MainWindow** class, locate the **ButtonCalculate_Click** method. This method runs when the user clicks the **Calculate** button.
4. In the **ButtonCalculate_Click** method, locate the line of code that invokes the **Matrix**.**MatrixMultiply** method, and enclose this line of code in a **try/catch** block that catches an **ArgumentException** exception named **ex**.
5. In the **catch** block, add a statement that displays a message box that contains the contents of the **Message** property of the exception object.

Hint: You can use the **MessageBox.Show** method to display a message box. Specify the message to display as a string passed in as a parameter to this method.

6. Build the solution and correct any errors.
7. Start the application without debugging.
8. In the MainWindow window, in the first drop-down list box, select **Matrix 1: 2 Columns**, in the second drop-down list box, select **Matrix 1: 2 Rows**, and then in the third drop-down list box, select **Matrix 2: 2 Columns**.

This creates a pair of 2 × 2 matrices initialized with zeroes.

9. Enter some non-negative values in the cells in both matrices, and then click **Calculate**.

Verify that the result is calculated and displayed, and that no exceptions occur.

10. Enter one or more negative values in the cells in either matrix, and then click **Calculate** again.

Verify that the appropriate exception message is displayed, and that it identifies the matrix and cell that is in error.

The application throws and catches exceptions, so you need to test that the application functions as expected. Although you can test for negative data points by using the application interface, the user interface does not let you create arrays of different dimensions. Therefore, you have been provided with unit test cases that will invoke the **MatrixMultiply** method with data that will cause exceptions. These tests have already been created; you will just run them to verify that your code works as expected.

- Task 4: Implement test cases and test the application

1. In the Matrix Unit Test Project, open the **MatrixTest** class, and then examine the **MatrixMultiplyTest1** method.

The **MatrixMultiplyTest1** method creates four matrices: **matrix1**, **matrix2**, **expected**, and **actual**. The **matrix1** and **matrix2** matrices are the input matrices that are passed to the **MatrixMultiply** method during the test. The **expected** matrix contains the expected result of the matrix multiplication, and the **actual** matrix stores the result of the **MatrixMultiply** method call. The method invokes the **MatrixMultiply** method before using a series of **Assert** statements to verify that the **expected** and **actual** matrices are identical.

This test method is complete and requires no further work.

2. Examine the **MatrixMultiplyTest2** method.

This method creates two compatible matrices, but **matrix2** contains a negative value. This should cause the **MatrixMultiply** method to throw an exception.

The **MatrixMultiplyTest2** method is prefixed with the **ExpectedException** attribute, indicating that the test method expects to cause an **ArgumentException** exception. If the test does not cause this exception, it will fail.

3. Examine the **MatrixMultiplyTest3** method.

This method creates two incompatible matrices and passes them to the **MatrixMultiply** method, which should throw an **ArgumentException** exception as a result. Again, the method is prefixed with the **ExpectedException** attribute, indicating that the test will fail if this exception is not thrown.

4. Run all tests in the solution, and verify that all tests execute correctly.

Задание 3. Использование Checked AnChecked для обработки переполнения целых чисел

In this exercise, you will examine what happens by default if an integer calculation causes numeric overflow. You will then modify the application to check for numeric overflow exceptions and repeat the calculation.

Scenario

Part of the software for a measuring device performs integer multiplication, but the integer values used can be very large. You want to ensure that the software does not generate errors that are caused by numeric overflow.

The main tasks for this exercise are as follows:

1. Open the IntegerOverflow solution.
2. Add a **checked** block.
3. Test the application.

- Task 1: Open the IntegerOverflow solution

The application multiplies 2147483647 by 2, and displays the result **–2**. This is because the multiplication causes an integer numeric overflow. By default, overflow errors of this nature do not cause an exception. However, in many situations, it is better to catch the overflow error than to let an application proceed with incorrect data.

In Visual Studio, on the **Debug** menu, click **Stop Debugging.**

- Task 2: Add a checked block

1. In Solution Explorer, open the MainWindow.xaml.cs file.
2. Locate the **DoMultiply_Click** method

This method runs when the user clicks the **Multiply** button.

3. Remove the **TODO - Place the multiplication in a checked block** comment. Add a **try/catch** block around the line of code that performs the multiplication operation, and then catch the **OverflowException** exception.
4. Inside the **try** block, add a **checked** block around the line of code that performs the multiplication arithmetic.
5. Build the solution and correct any errors.

- Task 3: Test the application

1. Start the application.
2. Click **Multiply**. Verify that the application now displays a message informing you that the arithmetic operation resulted in an overflow.
3. Click **OK**, close the MainWindow window, and then return to Visual Studio.

Задание 4. Подпишите отчет к работе №3 хэшем строки, состоящей из вашей фамилии, имени и отчества, написанных через запятые

1. Используйте алгоритм MD5 или (и) SHA256.
2. Подключите пространство имен using System.Security.Cryptography;
3. Преобразуйте строку ФИО в массив байтов (Encoding...);
4. Получите хэш в виде массива байтов (ComputeHash);
5. Преобразуйте хэш из массива в строку, состоящую из шестнадцатеричных символов (ToHexString());
6. Не забывайте про кодировку UTF8! C# использует Unicode!

Практическая работа №4.

Тема: Типы перечислений и структуры

Цель работы: Научиться на практике создавать перечисления и структуры. Разобраться самостоятельно с эффективностью использования структур, недостатки и преимущества по сравнению с классами

Ключевые понятия: Enum, Struct, Nulable.

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентацию лектора курса: «Классы, структуры, конструкторы, модификаторы доступа в C# » (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка. (Приложение A, Комментарии к документации).

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы

Программный код для каждого из заданий с подробными комментариями

Обратите внимание:

1. В отличие от класса не обязательно вызывать конструктор для создания объекта структуры.

2. Начиная с C# 10 структуры могут перегружать конструкторы по умолчанию. При этом доступна инициализация полей значениями по умолчанию.

3. Начиная с C# 11 при определении конструктора структуры в нем не обязательно инициализировать все поля структуры.

Задание 1. Создание структуры для работы с комплексными числами.

1. Написать программу, в которой предусмотреть создание структуры для работы с комплексными числами.
2. В структуре предусмотреть вычисление модуля комплексного числа через отдельное свойство (свойство только для чтения или с помощью отдельного метода).
3. Перегрузить операторы сложения, вычитания, умножения т деления для структуры.
4. Перегрузить оператор ==
5. Перегрузить ToString.
6. Написать метод для преобразования комплексных чисел из обычной формы (прямоугольной) в экспоненциальную и обратно.
7. Для проверки метода вычислите (2+i)*(3-i) путем первода в экспоненциальную форму и обратно.

Задание 2. Создание структуры для работы с трехмерными векторами.

1. Написать программу, в которой предусмотреть создание структуры для работы с трехмерными векторами.

2. В структуре предусмотреть вычисление модуля вектора через отдельное свойство (свойство только для чтения или с помощью отдельного метода).
3. Перегрузить операторы сложения, вычитания, умножения т деления для структуры.
4. Перегрузить оператор ==
5. Перегрузить ToString.
6. Перегрузить оператор умножения вектора на вещественное число (слева и справа).
7. Перегрузить векторное произведение векторов.
8. Для проверки алгоритма вычислить силу Кориолиса, действующую в Симферополе на студента, массой 70кг, бегущего на юг со скоростью 7 м/с.

Задание 3. Использование перечислений.

In this exercise, you will define enumerations that represent different materials under stress (stainless steel, aluminum, reinforced concrete, and titanium) and the cross-section of the girders (I-Beam, Box, Z-Shaped, and C-Shaped). You will also define another enumeration called **TestResult** that represents the results of a stress test.

The main tasks for this exercise are as follows:

1. Open the Enumeration solution.
2. Add enumerations to the **StressTest** namespace.
3. Retrieve the enumeration values.
4. Display the selection results.
5. Test the solution.

- Task 1: Open the Enumerations solution

```
namespace StressTest
{
    /// <summary>
    /// Enumeration of girder material types
    /// </summary>
    public enum Material
    {
        StainlessSteel,
        Aluminium,
        ReinforcedConcrete,
        Composite,
        Titanium
    }
    /// <summary>
    /// Enumeration of girder cross-sections
    /// </summary>
    public enum CrossSection
    {
        IBeam,
        Box,
        ZShaped,
        CShaped
    }
    /// <summary>
    /// Enumeration of test results
    /// </summary>
    public enum TestResult
    {
        Pass,
        Fail
    }
}
```

- Task 2: Add enumerations to the StressTest namespace

1. Review the task list.

2. Locate the **TODO - Implement Material, CrossSection, and TestResult enumerations** task, and then double-click this task. This task is located in the StressTestType.cs file.

3. In the **StressTest** namespace, define a new enumeration named **Material**. The enumeration should have the following values:

   a. **StainlessSteel**
   b. **Aluminum**
   c. **ReinforcedConcrete**
   d. **Composite**
   e. **Titanium**

4. Below the **Material** enumeration, define a new enumeration named **CrossSection**. The enumeration should have the following values:

   a. **IBeam**
   b. **Box**
   c. **Zshaped**
   d. **CShaped**

5. Below the **CrossSection** enumeration, define a new enumeration named **TestResult**. The enumeration should have the following values:

   a. **Pass**
   b. **Fail**

6. Build the solution and correct any errors.

- Task 3: Retrieve the enumeration values

1. In the TestHarness project, display the MainWindow.xaml window.

The purpose of the TestHarness project is to enable you to display the values from each of the enumerations. When the application runs, the three lists are populated with the values that are defined for each of the enumerations. The user can select an item from each list, and the application will construct a string from the corresponding enumerations.

2. In the task list, locate the **TODO - Retrieve user selections from the UI** task, and then double-click this task. This task is located in the **MainWindow**.**xaml**.**cs** class.

3. Remove the comment, and add code to the **selectionChanged** method to perform the following tasks:

   a. Create a **Material** object called **selectedMaterial** and initialize it to the value of the **SelectedItem** property in the **materials** list box.
   b. Create a **CrossSection** object called **selectedCrossSection** and initialize it to the value of the **SelectedItem** property in the **crosssections** list box.
   c. Create a **TestResult** object called **selectedTestResult** and initialize it to the value of the **SelectedItem** property in the **testresults** list box.

Hint: The **SelectedItem** property of a **ListBox** control has the **object** type. You must cast this property to the appropriate type when you assign it to an enumeration variable.

- Task 4: Display the selection results

1. In the **selectionChanged** method, after the code that you added in the previous task, add a statement to create a new **StringBuilder** object named **selectionStringBuilder**.

2. Add a **switch** statement to evaluate the selectedMaterial variable. In the **switch** statement, add **case** statements for each potential value of the **Material** enumeration. In each **case** statement, add code to append the text "Material: *<selectedMaterial>*, " to the **selectionStringBuilder** object. Substitute the text "*<selectedMaterial>*" in this string with the corresponding value for the selectedMaterial variable that is shown in the following table.

Material enumeration value	*<selectedMaterial>* string
Material.StainlessSteel	Stainless Steel
Material.Aluminum	Aluminum

Material.ReinforcedConcrete	Reinforced Concrete
Material.Composite	Composite
Material.Titanium	Titanium

3. Add another **switch** statement to evaluate the selectedCrossSection variable. In this **switch** statement, add **case** statements for each potential value of the **CrossSection** enumeration. In each **case** statement, add code to append the text "Cross-section: <*selectedCrossSection*>," to the **selectionStringBuilder** object. Substitute the text "<*selectedCrossSection*>" in this string with the corresponding value for the selectedCrossSection variable that is shown in the following table.

Material enumeration value	<*selectedCrossSection*> string
CrossSection.**IBeam**	I-Beam
CrossSection.**Box**	Box
CrossSection.**ZShaped**	Z-Shaped
CrossSection.**CShaped**	C-Shaped

4. Add a final **switch** statement to evaluate the **selectedTestResult** member. In the **switch** statement, add **case** statements for each potential value of the **TestResult** enumeration. In each **case** statement, add code to append the text "Result: <*selectedTestResult*>." to the **selectionStringBuilder** object. Substitute the text "<*selectedTestResult*>" in this string with the corresponding value for the selectedTestResult variable that is shown in the following table.

Material enumeration value	<*selectedTestResult*> string
TestResult.Pass	Pass
TestResult.Fail	Fail

5. At the end of the **selectionChanged** method, add code to display the string that is constructed by using the **selectionStringBuilder** object in the **Content** property of the **testDetails** label.

- Task 5: Test the solution

1. Build the application and correct any errors.
2. Run the application.
3. In the MainWindow window, in the **Material** list, click **Titanium**, in the **CrossSection** list, click **Box**, and then in the **Result** list, click **Fail**.

At the bottom of the window, verify that the label updates with your selections.

4. Experiment by selecting further values from all three lists, and verify that with each change, the label updates to reflect the changes.
5. Close the application, and then return to Visual Studio.

Задание 4. Использование структур

In this exercise, you will define a type called **TestCaseResult** that holds the result of a stress test. It will have the following public fields:

- **Result** : **TestResult**
- **ReasonForFailure**: **string**

This type is small, so it is best implemented as a struct. You will provide a constructor that initializes these fields.

The main tasks for this exercise are as follows:

1. Open the Structures solution.
2. Add the **TestCaseResult** structure.
3. Add an array of **TestCaseResult** objects to the user interface project.
4. Fill the **results** array with data.
5. Display the array contents.
6. Test the solution.

- Task 1: Open the Structures solution

```
namespace StressTest
{
    // Enumerations Exercise 1
    /// <summary>
    /// Enumeration of girder material types
    /// </summary>
public enum Material {StainlessSteel, Aluminium,
ReinforcedConcrete,Composite,Titanium }
    /// <summary>
    /// Enumeration of girder cross-sections
    /// </summary>
    public enum CrossSection { IBeam, Box, ZShaped, CShaped }
    /// <summary>
    /// Enumeration of test results
    /// </summary>
    public enum TestResult { Pass, Fail }
    // Structures Exercise 2
    /// <summary>
    /// Structure containing test results
    /// </summary>
    public struct TestCaseResult
    {
        /// <summary>
        /// Test result (enumeration type)
        /// </summary>
        public TestResult Result;
        /// <summary>
        /// Description of reason for failure
        /// </summary>
        public string ReasonForFailure;
    }
}
```

- Task 2: Add the TestCaseResult structure

1. Review the task list:
2. In the task list, locate the **TODO - Declare a Structure** task, and then double-click this task. This task is located in the StressTestTypes.cs file.
3. Delete the comment, and then declare a new structure named **TestCaseResult**. In the **TestCaseResult** structure, add the following members:
   a. A **TestResult** object named **Result**.
   b. A **string** object named **ReasonForFailure**.

- Task 3: Add an array of TestCaseResult objects to the user interface project

1. In the TestHarness project, display the MainWindow.xaml window.

This project simulates running stress tests and displays the results. It tracks the number of successful and failed tests, and for each failed test, it displays the reason for the failure.

2. In the task list, locate the **TODO - Declare a TestCaseResult array** task, and then double-click this task.

3. Remove the comment, and then declare a new array of **TestCaseResult** objects named **results**.

- Task 4: Fill the results array with data

1. In the **RunTests_Click** method, after the statement that clears the **reasonsList** list, add code to initialize the **results** array. Set the array length to **10**.
2. Below the statement that creates the array, add code that iterates through the items in the array and populates each one with the value that the static **GenerateResult** method of the **TestManager** class returns. The **GenerateResult** method simulates running a stress test and returns a **TestCaseResult** object that contains the result of the test and the reason for any failure.

- Task 5: Display the array contents
- Locate the comment **TODO - Display the TestCaseResult data**. Delete the comment, and then add code that iterates through the **results** array. For each value in the array, perform the following tasks:

a. Evaluate the **result** value. If the **result** value is **TestResult**.**Pass**, increment the **passCount** value.

b. If the **result** value is **TestResult.Fail**, increment the **failCount** value, and add the **ReasonForFailure** string to the **reasonsList** list box that is displayed in the window.

Note: To add an item to a list box, you use the **ListBox**.**Items**.**Add** method and pass the item to add to the list as a parameter to the method.

- Task 6: Test the solution

1. Build the application and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Tests**.

Verify that the **Successes** and **Failures** messages are displayed. Also verify that a message appears in the **Failures** list if failures occur.

4. Click **Run Tests** again to simulate running another batch of tests and display the results of these tests.
5. Close the application, and then return to Visual Studio.

Практическая работа №5.

Тема: Наследование классов и использование интерфейсов

Цель работы: Научиться на практике наследовать классы и интерфейсы. Разобраться на практике с вопросами отличия интерфейсов и абстрактных классов, реализовать множественное наследование интерфейсов и явную реализация интерфейсов. Научиться строить архитектуру приложения с помощью интерфейсов, разобраться с преимуществами такого подхода при промышленном программировании.

Ключевые понятия: Interface, Abstract Class, множественное наследование интерфейсов, явная реализация интерфейсов.

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентацию лектора курса: «Интерфейсы» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка.

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы
3. Программный код для каждого из заданий с подробными комментариями

Важно:

1. **Интерфейсы помогают определить взаимодействие между модулями программы, даже если эти модули ещё не существуют.**
2. **Использование интерфейсов позволяет обойти ограничения множественного наследования.**
3. **Интерфейсы обеспечивают изоляцию (слабую связанность) классов.**
4. **Интерфейсы упрощают тестирование программы.**
5. **Интерфейсы обеспечивают стандартизацию программирования, например все коллекции, списки, очереди, стек, и т.п. реализуют интерфейсы ICollection, IComparer, IEnumerable, IEnumerator, словари реализуют интерфейс IDictionary.**
6. **Обеспечивают реализацию поведенческих паттернов проектирования.**

Задание 1. Использование интерфейсов

In this exercise, you will define the following enumeration:

- Units: Metric, Imperial

You will then define a class called **MeasureLengthDevice** that implements the **IMeasuringDevice** interface and drives a device that measures the length of an object. This class will also include the following private fields:

- **unitsToUse**: Units
- **dataCaptured**: integer array
- **mostRecentMeasure**: integer

You will provide a constructor to initialize the fields in the class (the user will specify a parameter that populates **unitsToUse**).

When the device starts running (when the **StartCollecting** method is called), the device will capture data and store it in the **dataCaptured** array (you will simulate this in the lab by using the code that is provided). This array has a finite, fixed size; when the device is full, it will wrap around and start to overwrite the oldest data. Each time that it takes a new measurement, the device copies this measurement to the **mostRecentMeasure** field. The **GetRawData** method will return the contents of the array. The **MetricValue** and **ImperialValue** methods will return the value in this field, converted according to the units that are specified in the **unitsToUse** field. If **unitsToUse** is **Metric**, **MetricValue** simply returns the data and **ImperialValue** performs a calculation to convert the data to imperial units. Similarly, if **unitsToUse** is **Imperial**, **ImperialValue** simply returns the data and **MetricValue** performs a calculation to convert the data to metric units.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create the **Units** enumeration.
3. Create the **MeasureLengthDevice** class.
4. Update the test harness.
5. Test the **MeasureLengthDevice** class by using the test harness.

- Task 1: Open

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
        /// <summary>
/// Converts the raw data collected by the measuring device into a metric value
        /// </summary>
///<returns>The latest measurement from the device converted to metric units.</returns>
        decimal MetricValue();
        /// <summary>
  /// Converts the raw data collected by the measuring device into an imperial value.
        /// </summary>
///<returns>The latest measurement from the device converted to imperial
units.</returns>
        decimal ImperialValue();
        /// <summary>
        /// Starts the measuring device.
        /// </summary>
        void StartCollecting();
        /// <summary>
        /// Stops the measuring device.
        /// </summary>
        void StopCollecting();
        /// <summary>
/// Enables access to the raw data from the device in whatever units are native to the device
        /// </summary>
/// <returns>The raw data from the device in native format.</returns>
        int[] GetRawData();
    }
      }
```

- Task 2: Create the Units enumeration

1. In Visual Studio, review the task list.
2. In the task list, double-click the task **TODO: Implement the Units enumeration**. This task is located in the UnitsEnumeration.cs file.
3. Remove the TODO comment in the UnitsEnumeration file and declare an enumeration named **Units**. The enumeration must be accessible from code in different assemblies.
4. Add the values **Metric** and **Imperial** to the enumeration.
5. Comment your code to make it easier for developers who use the enumeration.

6. Build the solution and correct any errors.

- Task 3: Create the MeasureLengthDevice class

1. In the task list, double-click the task **TODO: Implement the MeasureLengthDevice class**. This task is located in the MeasureLengthDevice.cs file.
2. Remove the TODO comment and add a **public** class named **MeasureLengthDevice**.
3. Modify the **MeasureLengthDevice** class declaration to implement the **IMeasuringDevice** interface.
4. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.
5. Bring the **DeviceControl** namespace into scope.

The MeasuringDevice project already contains a reference to the DeviceController project. You are writing code to control a device. However, because the physical device is not available with this lab, the DeviceController project enables you to call methods that control an emulated device. The DeviceController project does not include a visual interface; to control the device, you must use the classes and methods that the project exposes. The DeviceController project is provided complete. You can review the code if you wish, but you do not need to modify it.

6. After the method stubs that the Implement Interface Wizard added in the **MeasureLengthDevice** class, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

DeviceType is an enumeration that contains the values **LENGTH** and **MASS**. It is used to specify the type of measurement that the device records. It is defined in the DeviceController project.

7. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.LENGTH**.
8. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to the **StartCollecting** method to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the value in the **measurementType** field as the parameter to the **StartCollecting** method.
9. In the **StartCollecting** method, call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.
10. Add the **GetMeasurements** method to the class, as shown in the following code example.

Note: A code snippet is available, called **GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();

        while (controller != null)
        {
```

```
                System.Threading.Thread.Sleep(timer.Next(1000, 5000));
                dataCaptured[x] = controller != null ?
                    controller.TakeMeasurement() : dataCaptured[x];
                mostRecentMeasure = dataCaptured[x];
                x++;
                if (x == 10)
                {
                    x = 0;
                }
            }
        });
    }
```

The **GetMeasurements** method retrieves measurements from the emulated device. In this module, you will use the code in the **GetMeasurements** method to populate the **dataCaptured** array. This array acts as a fixed-length circular buffer, overwriting the oldest value each time a new measurement is taken. In a later module, you will modify this class to respond to events that the device raises whenever it detects a new measurement.

11. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.
12. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.
13. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
14. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 25.4.
15. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 0.03937.
16. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
17. Build the solution and correct any errors.

- Task 4: Update the test harness

The test harness application for this lab is a simple Windows® Presentation Foundation (WPF) application that is designed to test the functionality of the **MeasureLengthDevice** class that you have just developed. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file by clicking the first **TODO: Add code to instantiate the device field** item in the task list. This task is located in the **createInstance_Click** method in the WPF window, and it runs when the user clicks the **Create Instance** button.
3. In the **createInstance_Click** method, replace both TODO comments with code to instantiate a field called **device** and set it to an instance of the **MeasureLengthDevice** class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureLengthDevice** constructor.
4. Build the solution and correct any errors.

- Task 5: Test the MeasureLengthDevice class by using the test harness

1. Set the Exercise2TestHarness project to be the default startup project.
2. Start the Exercise2TestHarness application.

3. Choose **Imperial**, and then click **Create MeasureLengthDevice Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.
4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **device** object that the **IMeasuringDevice** interface defines.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Raw Data**. You should see up to 10 values in the list box in the lower part of the window. This is the data that the device emulator has generated. It is stored in the **dataCaptured** array by the **GetMeasurements** method in the **MeasureLengthDevice** class. The **dataCaptured** array acts as a fixed-length circular buffer. Initially, it contains zero values, but as the device emulator reports measurements, they are added to this array. When the array is full, it wraps around and starts overwriting data, beginning with the oldest measurement.
7. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement. Note that a new measurement might have been taken since you clicked the **Get Raw Data** button.
8. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
9. Click **Stop Collecting**.
10. Choose **Metric**, and then click **Create MeasureLengthDevice Instance**. This action creates a new instance of the device emulator that uses metric measurements.
11. Click **Start Collecting**. This button starts the new **device** object.
12. Wait for 10 seconds.
13. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
14. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
15. Click **Stop Collecting**.
16. Close the Exercise 2 Test Harness window.

Задание 2. Создание абстрактного класса

In this exercise, you will define a class called **MeasureMassDevice**, which also implements the **IMeasuringDevice** interface. You will notice that, although the **MetricValue** and **ImperialValue** methods are implemented slightly differently from the **MeasureLength** class, the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods are identical. Code duplication is never a good thing, and can lead to maintenance difficulties. Consequently, you will create an abstract class called **MeasureDataDevice** that provides default implementations of the duplicated methods. Students will modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from this class.

The main tasks in this exercise are as follows:

1. Open the starter project.
2. Create the **MeasureMassDevice** class.
3. Update the test harness.
4. Test the **MeasureMassDevice** class by using the test harness.
5. Create the **MeasureDataDevice** abstract class.
6. Modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from the **MeasureDataDevice** abstract class.
7. Test the classes by using the test harness.

- Task 1: Open the starter project

```
namespace MeasuringDevice
{
    public abstract class MeasureDataDevice : IMeasuringDevice
    {
        /// <summary>
```

```
    /// Converts the raw data collected by the measuring device into a metric value.
        /// </summary>
/// <returns>The latest measurement from the device converted to metric units.</returns>
        public abstract decimal MetricValue();
        /// <summary>
 /// Converts the raw data collected by the measuring device into an imperial value.
        /// </summary>
///<returns>The latest measurement from the device converted to imperial units.</returns>
        public abstract decimal ImperialValue();
        /// <summary>
        /// Starts the measuring device.
        /// </summary>
        public void StartCollecting()
        {
            controller = DeviceController.StartDevice(measurementType);
            GetMeasurements();
        }
        /// <summary>
        /// Stops the measuring device.
        /// </summary>
        public void StopCollecting()
        {
            if (controller != null)
            {
                controller.StopDevice();
                controller = null;
            }
        }
```

- Task 2: Create the MeasureMassDevice class

1. In Visual Studio, review the task list.
2. Open the MeasureMassDevice.cs file.
3. Replace the TODO comment with a **public** class named **MeasureMassDevice**.
4. Modify the **MeasureMassDevice** class declaration to implement the **IMeasuringDevice** interface.
5. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.
6. Bring the **DeviceControl** namespace into scope.

The MeasuringDevice project already contains a reference to the DeviceController project. This project implements the **DeviceController** type, which provides access to the measuring device emulator.

7. After the method stubs that Visual Studio added, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

8. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.MASS**.
9. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to instantiate the **controller** field by using the static **StartDevice** method of the

DeviceController class. Pass the **measurementType** field as the parameter to the **StartDevice** method.

10. Add code to call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.

11. Add the **GetMeasurements** method to the class, as shown in the following code example.

Note1: class DeviceController:

```
public class DeviceController : IDisposable
    {
        private IControllableDevice device;

        /// <summary>
        /// A factory method to create a start a new instance of a device.
        /// </summary>
        /// <param name="MeasurementType">Specifies which type of device to start.
Must be MASS or LENGTH.</param>
        /// <returns>An instance of the DeviceController class with the controlled
device in the started state.</returns>
        public static DeviceController StartDevice(DeviceType MeasurementType)
        {
            DeviceController controller = new DeviceController();
            switch (MeasurementType)
            {
                case DeviceType.LENGTH:
                    controller.device = new FabrikamDevices.LengthMeasuringDevice();
                    break;
                case DeviceType.MASS:
                    controller.device = new ContosoDevices.MassMeasuringDevice();
                    break;
            }
            if (controller.device != null)
            {
                controller.device.StartDevice();
            }

            return controller;
        }

        /// <summary>
        /// Stops the controlled device.
        /// </summary>
        public void StopDevice()
        {
            device.StopDevice();
        }

        /// <summary>
        /// Forces the controlled device to record a measurement.
        /// </summary>
        /// <returns>The measurement taken by the device.</returns>
        public int TakeMeasurement()
        {
            return device.GetLatestMeasure();
        }

        /// <summary>
        /// Disposes the device.
        /// </summary>
        public void Dispose()
        {
        }
```

```
    }
```

Note2: metod GetMeasurements:

A code snippet is available, called **Mod8GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();
        while (controller != null)
        {
            System.Threading.Thread.Sleep(timer.Next(1000, 5000));
            dataCaptured[x] = controller != null ?
                controller.TakeMeasurement() : dataCaptured[x];
            mostRecentMeasure = dataCaptured[x];
            x++;
            if (x == 10)
            {
                x = 0;
            }
        }
    });
}
```

This is the same method that you defined for the **MeasureLengthDevice** class.

12. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.
13. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.
14. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
15. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 0.4536.
16. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 2.2046.
17. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
18. Build the solution and correct any errors.

- Task 3: Update the test harness

The test harness application in this lab is a modified version of the WPF application that you used in Exercise 2. It is designed to test the functionality of the **MeasureLengthDevice** and **MeasureMassDevice** classes. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file by using the first **TODO: Instantiate the device field by using the new MeasureMassDevice class** item in the task list.
3. In the **createInstance_Click** method, replace both TODO comments with code to instantiate the device field to an instance of the **MeasureMassDevice** class. You

must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureMassDevice** constructor.

4. Build the solution and correct any errors.

- Task 4: Test the MeasureMassDevice class by using the test harness

1. Set the Exercise3TestHarness project to be the default startup project.
2. Start the Exercise3TestHarness application.
3. Choose **Imperial**, choose **Mass Device**, and then click **Create Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.
4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **MeasureMassDevice** object.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement.
7. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
8. Click **Stop Collecting**.
9. Choose **Metric**, and then click **Create Instance**. This action creates a new instance of the device emulator that uses metric measurements.
10. Click **Start Collecting**. This button starts the new **device** object.
11. Wait for 10 seconds.
12. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
13. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
14. Click **Stop Collecting**.

- Task 5: Create the MeasureDataDevice abstract class

You have developed two classes, **MeasureLengthDevice** and **MeasureMassDevice**. Much of the functionality of these classes is common to both. This code duplication is unnecessary and risks introducing bugs. To reduce the code that is required and the risk of introducing bugs, you will create an abstract class that will contain the common functionality.

1. Open the MeasureDataDevice.cs file.
2. Remove the TODO comment and add an **abstract** class named **MeasureDataDevice**.
3. Modify the **MeasureDataDevice** class declaration to implement the **IMeasuringDevice** interface.
4. Bring the **DeviceControl** namespace into scope.
5. In the **MeasureDataDevice** class, add a **public abstract** method named **MetricValue**. This method should return a **decimal** value, but not take any parameters.

The implementation of the **MetricValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes. Declaring the **MetricValue** method as **abstract** forces child classes to implement this method.

Hint: Look at the code for the **MetricValue** method for the **MeasureLengthDevice** and **MeasureMassDevice** classes. You will observe that they are quite similar, apart from the conversion factors that are used, and you could factor this logic out into a method in the abstract **MeasureDataDevice** class. However, for the sake of this exercise, assume that these methods are totally different. The same note applies to the **ImperialValue** method that you will define in the next step.

6. In the **MeasureDataDevice** class, add a **public abstract** method with a **decimal** return type named **ImperialValue**.

Like the **MetricValue** method, the implementation of the **ImperialValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes.

7. In the MeasureLengthDevice.cs file, locate and copy the code for the **StartCollecting** method, and then add this method to the **MeasureDataDevice** class.

Visual Studio will warn you that the controller variable, the **measurementType** enumeration, and the **GetMeasurements** method are not defined. You will add these items to the **MeasureDataDevice** class in later steps in this task.

8. Copy the **StopCollecting** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

Visual Studio will warn you that the controller variable is not defined.

9. Copy the **GetRawData** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

Visual Studio will warn you that the dataCaptured variable is not defined.

10. Copy the **GetMeasurements** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

Visual Studio will warn you that the dataCaptured, controller, and mostRecentMeasure variables are not defined.

11. Copy the five fields in the following table from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

The warnings in the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods should disappear.

12. In the **MeasureDataDevice** class, modify the five fields that you added in the previous step to make them visible to classes that inherit from the abstract class.

13. Modify the declaration of the **measurementType** field so that it is no longer constant and not instantiated when it is declared.

14. Build the solution and correct any errors.

- Task 6: Modify the MeasureLengthDevice and MeasureMassDevice classes to inherit from the MeasureDataDevice abstract class

In this task, you will remove the duplicated code from the **MeasureLengthDevice** and **MeasureMassDevice** classes by modifying them to inherit from the **MeasureDataDevice** abstract class that you created in the previous task.

1. In the MeasureLengthDevice.cs file, modify the declaration of the **MeasureLengthDevice** class so that, in addition to implementing the **IMeasuringDevice** interface, it also inherits from the **MeasureDataDevice** class.
2. Remove the **StartCollecting** method from the **MeasureLengthDevice** class.
3. Remove the **StopCollecting** method from the **MeasureLengthDevice** class.
4. Remove the **GetRawData** method from the **MeasureLengthDevice** class.
5. Remove the **GetMeasurements** method from the **MeasureLengthDevice** class.
6. Remove the fields in the following table from the **MeasureLengthDevice** class.

Name	Type	Accessor
unitsToUse	Units	private

dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

7. Modify the constructor to set the **measurementType** field to **DeviceType.LENGTH**.
8. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
9. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
10. In the MeasureMassDevice.cs file, modify the declaration of the **MeasureMassDevice** class so that it inherits from the **MeasureDataDevice** class.
11. Remove the **StartCollecting** method from the **MeasureMassDevice** class.
12. Remove the **StopCollecting** method from the **MeasureMassDevice** class.
13. Remove the **GetRawData** method from the **MeasureMassDevice** class.
14. Remove the **GetMeasurements** method from the **MeasureMassDevice** class.
15. Remove the fields in the following table from the **MeasureMassDevice** class.

Name	Type	Accessor	
unitsToUse	Units		private
dataCaptured	int[]		private
mostRecentMeasure	int		private
controller	DeviceController		private
measurementType	DeviceType		private

16. Modify the constructor to set the **measurementType** field to **DeviceType.MASS**.
17. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
18. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
19. Build the solution and correct any errors.

Задание 3. Создание архитектуры приложения с помощью интерфейсов

1. Изучить возможность и преимущества создания архитектуры приложения с помощью интерфейсов.
2. Написать в отчет о работе краткую справку об использовании абстрактных классов и интерфейсов при проектировании крупных промышленных приложений. Советуем посмотреть видеоурок https://www.youtube.com/watch?v=eXapyrhtj7I&t=111s
3. Создать архитектуру собственного приложения с помощью интерфейсов. Обеспечить частичную или полную реализацию классов. В качестве примера можно выбрать: книжный магазин, расширенный блокнот для делопроизводства, интернет магазин для продажи компьютеров и т.п.
4. Разработать и включить в отчет диаграмму UML для вашего приложения.

Практическая работа №6

Тема: Делегаты, события, лямбда выражения, многопоточное и асинхронное программирование

Цель работы: Научиться на практике использованию делегатов, лямбда-выражений и событий. Научиться создавать собственные события.

Ключевые понятия: delegate; MulticastDelegate ; функции обратного вызова (callback); цепочки делегатов “+=”; “-=”; лябда-выражения “=>”; Event; EventHandler; класс Sender и классы Receivers; класс EventArgs и его потомки, Thread, Lock, Join, Wait, Pulse, Mutex, Semaphore, TryEnter, Priority, Background, Start, IsAlive, Barrier, TPL, Task, Parallel, Async, Await.

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентации лектора курса: «Делегаты», «События», «Многопоточное программирование» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка. (Приложение A, Комментарии к документации).

Содержание отчета:

1. Описание и раскрытие ключевых понятий
2. Краткое описание изученной литературы
3. Программный код для каждого из заданий с подробными комментариями

Задание 1. Использование событий

In this exercise, you will modify the **IMeasuringDevice** interface and add an event called **NewMeasurementTaken**. This event will be triggered whenever the device detects a change and takes a new measurement.

You will modify the **MeasureDataDevice** abstract class from the previous lab and implement this event. The **NewMeasurementTaken** event will occur after the device has populated the internal buffer with the new measurement and logged it.

You will use a **BackgroundWorker** component to poll for new measurements. The polling for new measurements will take place in the **DoWork** event, and the **ProgressReported** event will raise the **NewMeasurementTaken** event to notify the client application that a new measurement has been taken.

You will start the background thread running by using the **RunWorkerAsync** method, and the device will support cancellation of the background thread by using the **CancelWorkerAsync** method.

You will test the new functionality by using an existing WPF application that creates an instance of the **MeasureMassDevice** class and trapping the events that it raises by using a delegate. The WPF application should be able to pause and then restart the the **MeasureMassDevice** class.

The main tasks for this exercise are as follows:

1. Open the Events solution.
2. Create a new interface that extends the **IMeasuringDevice** interface.

3. Add the **NewMeasurementTaken** event to the **MeasureDataDevice** class.
4. Add a **BackgroundWorker** member to the **MeasureDataDevice** class.
5. Add the **GetMeasurements** method to the **MeasureDataDevice** class.
6. Implement the **dataCollector_DoWork** method.
7. Implement the **dataCollector_ProgressChanged** method.
8. Call the **GetMeasurements** method to start collecting measurements.
9. Call the **CancelAsync** method to stop collecting measurements.
10. Dispose of the **BackgroundWorker** object when the **MeasureDataDevice** object is destroyed.
11. Update the UI to handle measurement events.
12. Implement the **device_NewMeasurementTaken** event-handling method.
13. Disconnect the event handler.
14. Test the solution.

- Task 1: Open the Events solution

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
        /// <summary>
   /// Converts the raw data collected by the measuring device into a metric
value.
        /// </summary>
    /// <returns>The latest measurement from the device converted to metric
units.</returns>
        decimal MetricValue();
        /// <summary>
    /// Converts the raw data collected by the measuring device into an imperial
value.
        /// </summary>
  /// <returns>The latest measurement from the device converted to imperial
units.</returns>
        decimal ImperialValue();
        /// <summary>
        /// Starts the measuring device.
        /// </summary>
        void StartCollecting();
        /// <summary>
        /// Stops the measuring device.
        /// </summary>
        void StopCollecting();
        /// <summary
  /// Enables access to the raw data from the device in whatever units are native to the
device
        /// </summary>
        /// <returns>The raw data from the device in native format.</returns>
        int[] GetRawData();
        /// <summary>
        /// Returns the file name of the logging file for the device.
        /// </summary>
        /// <returns>The file name of the logging file.</returns>
        string GetLoggingFile();
        /// <summary>
        /// Gets the Units used natively by the device.
        /// </summary>
        Units UnitsToUse { get; }
        /// <summary>
        /// Gets an array of the measurements taken by the device.
        /// </summary>
        int[] DataCaptured { get; }
        /// <summary>
```

```
            /// Gets the most recent measurement taken by the device.
            /// </summary>
            int MostRecentMeasure { get; }
            /// <summary>
            /// Gets or sets the name of the logging file used.
        /// If the logging file changes this closes the current file and creates the
new file
            /// </summary>
            string LoggingFileName { get; set; }
        }
    }
```

- Task 2: Create a new interface that extends the IMeasuringDevice interface

1. In the MeasuringDevice project, add a new interface named **IEventEnabledMeasuringDevice** in a file named IEventEnabledMeasuringDevice.cs.

```
    namespace MeasuringDevice
    {
        interface IEventEnabledMeasuringDevice : IMeasuringDevice
        {
            event EventHandler NewMeasurementTaken;
            // Event that fires every heartbeat.
            event HeartBeatEventHandler HeartBeat;
            // Read only heartbeat interval - set in constructor.
            int HeartBeatInterval { get; }
        }
    }
```

Note: Creating a new interface that extends an existing interface is good programming practice, because it preserves the structure of the original interface for backward compatibility with preexisting code. All preexisting code can reference the original interface, and new code can reference the new interface and take advantage of any new functionality.

- Modify the interface definition so that the **IEventEnabledMeasuringDevice** interface extends the **IMeasuringDevice** interface.

3. In the **IEventEnabledMeasuringDevice** interface, add an event named **NewMeasurementTaken** by using the base **EventHandler** delegate.
4. Build the application to enable Microsoft IntelliSense® to reflect your changes.

- Task 3: Add the NewMeasurementTaken event to the MeasureDataDevice class

1. Review the task list.
2. Locate the **TODO - Modify the class definition to implement the extended interface** task, and then double-click this task. This task is located in the MeasureDataDevice class file.
3. Remove the **TODO - Modify the class definition to implement the extended interface** comment, and then modify the class definition to implement the **IEventEnabledMeasuringDevice** interface instead of the **IMeasuringDevice** interface.
4. In the task list, locate the **TODO - Add the NewMeasurementTaken event** task, and then double-click this task. This task is located at the end of the **MeasureDataDevice** class.
5. Remove the **TODO - Add the NewMeasurementTaken event** comment, and then declare an event named **NewMeasurementTaken** by using the same signature as the interface.
6. Below the event, remove the **TODO - Add an OnMeasurementTaken method** comment, and then add a protected virtual method named **OnNewMeasurementTaken**. The method should accept no parameters and have a void return type. The **MeasureDataDevice** class will use this method to raise the **NewMeasurementTaken** event.

7. In the **OnNewMeasurementTaken** method, add code to check that there is a subscriber for the **NewMeasurementTaken** event; if so, raise the event. The signature of the **EventHandler** delegate defines two parameters: an *object* parameter that indicates the object that raised the event and an *EventArgs* parameter that provides any additional data that is passed to the event handler. Set the *object* parameter to **this** and the *EventArgs* parameter to **null**.

Note: It is good programming practice to check that there are subscribers for an event before you raise it. If an event has no subscribers, the related delegate is null, and the .NET Framework runtime will throw an exception if the event is raised.

- Task 4: Add a BackgroundWorker member to the MeasureDataDevice class

1. In the task list, locate the **TODO - Declare a BackgroundWorker to generate data** task, and then double-click this task. This task is located near the top of the **MeasureDataDevice** class.
2. Remove the **TODO - Declare a BackgroundWorker to generate data** comment, and then add a private **BackgroundWorker** member named **dataCollector** to the class.

- Task 5: Add the GetMeasurements method to the MeasureDataDevice class

The **GetMeasurements** method will initialize the **dataCollector BackgroundWorker** member to poll for new measurements and raise the **NewMeasurementTaken** event each time it detects a new measurement.

- In the task list, locate the **TODO - Implement the GetMeasurements method** task, and then double-click this task.
- Remove the **TODO - Implement the GetMeasurements method** comment, and then add a new private method named **GetMeasurements** to the class. This method should take no parameters and not return a value.
- In the **GetMeasurements** method, add code to perform the following actions:

1. Instantiate the **dataCollector BackgroundWorker** member.
2. Specify that the **dataCollector BackgroundWorker** member supports cancellation.
3. Specify that the **dataCollector BackgroundWorker** member reports progress while running.

Hint: Set the **WorkerSupportsCancellation** and **WorkerReportsProgress** properties.

- Add the following code to instantiate a **DoWorkEventHandler** delegate that refers to a method called **dataCollector_DoWork**. Attach the delegate to the **DoWork** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_DoWork** method when the **DoWork** event is raised.

Hint: Use IntelliSense to generate a code stub for the **dataCollector_DoWork** method. To do this, type the first part of the line of code, up to the **+=** operators, and then press the TAB key twice. Visual Studio uses a built-in code snippet to complete the line of code and then add a method stub. You can do this each time you hook up an event handler to an event by using the **+=** compound assignment operator.

```
...
dataCollector.WorkerReportsProgress = true;
dataCollector.DoWork +=
    new DoWorkEventHandler(dataCollector_DoWork);
} ...
```

5. Using the same technique as in the previous step, instantiate a **ProgressChangedEventHandler** delegate that refers to a method called **dataCollector_ProgressChanged**. Attach this delegate to the **ProgressChanged** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_ProgressChanged** method when the **ProgressChanged** event is raised.

6. Add code to start the **dataCollector BackgroundWorker** object running asynchronously.

- Task 6: Implement the dataCollector_DoWork method

1. Underneath the **GetMeasurements** method, locate the **dataCollector_DoWork** method.

This method was generated during the previous task. It runs on a background thread, and its purpose is to collect and store measurement data.

- In the **dataCollector_DoWork** method, remove the statement that raises the **NotImplementedException** exception and add code to perform the following actions:

1. Instantiate the **dataCaptured** array with a new integer array that contains 10 items.
2. Define an integer i with an initial value of zero. You will use this variable to track the current position in the **dataCaptured** array.
3. Add a **while** loop that runs until the **dataCollector**.**CancellationPending** property is **false**.

- In the **while** loop, add code to perform the following actions:

1. Invoke the **controller**.**TakeMeasurement** method, and store the result in the **dataCaptured** array at the position that the integer **i** indicates. The **TakeMeasurement** method of the **controller** object blocks until a new measurement is available.
2. Update the **mostRecentCapture** property to contain the value in the **dataCaptured** array at the position that the integer **i** indicates.
3. If the value of the disposed variable is **true**, terminate the **while** loop. This step ensures that the measurement collection stops when the **MeasureDataDevice** object is destroyed.

- Add code to the **while** loop after the statements that you added in the previous step to perform the following actions:

1. Check whether the **loggingFileWriter** property is null.
2. If the **loggingFileWriter** property is not null, call the **loggingFileWriter**.**Writeline** method, passing a string parameter of the format "Measurement - *mostRecentMeasure*" where *mostRecentMeasure* is the value of the mostRecentMeasure variable.

Note: The **loggingFileWriter** property is a simple **StreamWriter** object that writes to a text file. This property is initialized in the **StartCollecting** method. You can use the **WriteLine** method to write to a **StreamWriter** object.

- Add a line of code to the end of the **while** loop to invoke the **dataCollector**.**ReportProgress** method, passing zero as the parameter.

The **ReportProgress** method raises the **ReportProgress** event and is normally used to return the percentage completion of the tasks assigned to the **BackgroundWorker** object. You can use the **ReportProgress** event to update progress bars or time estimates in the UI. In this case, because the task will run indefinitely until canceled, you will use the **ReportProgress** event as a mechanism to prompt the UI to refresh the display with the new measurement.

6. Add code to the end of the **while** loop to perform the following actions:

a. Increment the integer **i**.
b. If the value of the integer is greater than nine, reset **i** to zero.

You are using the integer i as a pointer to the next position to write to in the **dataCaptured** array. This array has space for 10 measurements. When element 9 is filled, the device will start to overwrite data beginning at element 0.

- Task 7: Implement the dataCollector_ProgressChanged method

Locate the **dataCollector_ProgressChanged** method.

This method was generated during an earlier task. It runs when the **ProgressChanged** event is raised. In this exercise, this event occurs when the **dataCollector_DoWork** method takes and stores a new measurement.

2. In the event handler, delete the exception code, and then invoke the **OnNewMeasurementTaken** method, passing no parameters.

The **OnNewMeasurementTaken** method raises the **NewMeasurementTaken** event that you defined earlier. You will modify the UI to subscribe to this event, so that when it is raised, the UI can update the displayed information.

- Task 8: Call the GetMeasurements method to start collecting measurements

1. In the task list, locate the **TODO - Call the GetMeasurements method** task, and then double-click this task. This task is located in the **StartCollecting** method.
2. Remove the **TODO - Call the GetMeasurements method** comment, and add a line of code to invoke the **GetMeasurements** method.

- Task 9: Call the CancelAsync method to stop collecting measurements

1. In the task list, locate the **TODO - Cancel the data collector** task, and then double-click this task. This task is located in the **StopCollecting** method.
2. Remove the **TODO - Cancel the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **CancelAsync** method to stop the work performed by the **dataCollector BackgroundWorker** object.

- Task 10: Dispose of the BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - Dispose of the data collector** task, and then double-click this task. This task is located in the **Dispose** method of the **MeasureDataDevice** class.
2. Remove the **TODO - Dispose of the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **Dispose** method to dispose of the **dataCollector** instance.

- Task 11: Update the UI to handle measurement events

1. In the task list, locate the **TODO - Declare a delegate to reference NewMeasurementEvent** task, and then double-click this task. This task is located in the code behind the MainWindow.xaml window.
2. Remove the comment and add code to define a delegate of type **EventHandler** named **newMeasurementTaken**.
3. In the **startCollecting_Click** method, remove the comment **TODO - use a delegate to refer to the event handler**, and add code to initialize the **newMeasurementTaken** delegate with a new **EventHandler** delegate that is based on a method named **device_NewMeasurementTaken**. You will create the **device_NewMeasurementTaken** method in the next task.

Note: You cannot use IntelliSense to automatically generate the stub for the **device_NewMeasurementTaken** method, as you did in earlier tasks.

- In the **startCollecting_Click** method, remove the **TODO - Hook up the event handler to the event** comment, and add code to connect the **newMeasurementTaken** delegate to the **NewMeasurementTaken** event of the **device** object. The **device** object is an instance of the **MeasureMassDevice** class, which inherits from the **MeasureDataDevice** abstract class.

Hint: To connect a delegate to an event, use the **+=** compound assignment operator on the event.

- Task 12: Implement the device_NewMeasurementTaken event-handling method

In the task list, locate the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** task, and then double-click this task.

Remove the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** comment, and add a private event-handler method named **device_NewMeasurementTaken**. The method should not return a value, but should take the following parameters:

An **object** object named **sender**.

An **EventArgs** object named **e**.

In the **device_NewMeasurementTaken** method, add code to check that the **device** member is not null. If the **device** member is not null, perform the following tasks:

Update the **Text** property of the **mostRecentMeasureBox** text box with the value of the **device.MostRecentMeasure** property.

Hint: Use the **ToString** method to convert the value that the **device.MostRecentMeasure** property returns from an integer to a string.

- Update the **Text** property of the **metricValueBox** text box with the value that the **device.MetricValue** method returns.
- Update the **Text** property of the **imperialValueBox** text box with the value that the **device.ImperialValue** method returns.
- Reset the **rawDataValues.ItemsSource** property to **null**.
- Set the **rawDataValues.ItemsSource** property to the value that the **device.GetRawData** method returns.

Note: The final two steps are both necessary to ensure that the data-binding mechanism that the **Raw Data** box uses on the WPF window updates the display correctly.

- Task 13: Disconnect the event handler

In the task list, locate the **TODO - Disconnect the event handler** task, and then double-click this task. This task is located in the **stopCollecting_Click** method, which runs when the user clicks the **Stop Collecting** button.

Remove the **TODO - Disconnect the event handler** comment, and add code to disconnect the **newMeasurementTaken** delegate from the **device.NewMeasurementTaken** event.

Hint: To disconnect a delegate from an event, use the **-=** compound assignment operator on the event.

- Task 14: Test the solution

Build the project and correct any errors.

Start the application.

Click **Start Collecting**, and verify that measurement values begin to appear in the **Raw Data** box.

The **MeasureMassDevice** object used by the application takes metric measurements and stores them, before raising the **NewMeasurementTaken** event. The event calls code that updates the UI with the latest information. Continue to watch the **Raw Data** list box to see the buffer fill with data and then begin to overwrite earlier values.

- Click **Stop Collecting**, and verify that the UI no longer updates.
- Click **Start Collecting** again. Verify that the **Raw Data** list box is cleared and that new measurement data is captured and displayed.
- Click **Stop Collecting**.
- Close the application, and then return to Visual Studio.

Задание 2. Использование лямбда-выражений

In this exercise, you will declare a new delegate type and a new **EventArgs** type to support the **HeartBeat** event. You will modify the **IMeasuringDevice** interface and the **MeasureDataDevice** class to generate the heartbeat by using a **BackgroundWorker** object. You will specify the code to run on the new thread by using a lambda expression.

In the **ReportProgress** event handler, you will specify the code to notify the client application with another lambda expression.

You will handle the **HeartBeat** event in the WPF application by using a lambda expression.

The main tasks for this exercise are as follows:

1. Open the Events solution.
2. Define a new **EventArgs** class to support heartbeat events.
3. Declare a new delegate type.
4. Update the **IEventEnabledMeasuringDevice** interface.
5. Add the **HeartBeat** event and **HeartBeatInterval** property to the **MeasureDataDevice** class.
6. Use a **BackgroundWorker** object to generate the heartbeat.
7. Call the **StartHeartBeat** method when the **MeasureDataDevice** object starts running.
8. Dispose of the **heartBeatTimer BackgroundWorker** object when the **MeasureDataDevice** object is destroyed.
9. Update the constructor for the **MeasureMassDevice** class.
10. Handle the **HeartBeat** event in the UI.
11. Test the solution.

- Task 1: Open the Events solution
- Open the Events solution in the.

Note: The Events solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1; however, it includes an updated task list to enable you to complete this exercise.

- Task 2: Define a new EventArgs class to support heartbeat events
- In the MeasuringDevice project, add a new code file named HeartBeatEvent.cs.
- In the code file, add a **using** directive to bring the **System** namespace into scope.
- Define a new class named **HeartBeatEventArgs** in the **MeasuringDevice** namespace. The class should extend the **EventArgs** class.

Note: A custom event arguments class can contain any number of properties; these properties store information when the event is raised, enabling an event handler to receive event-specific information when the event is handled.

- In the **HeartBeatEventArgs** class, add a read-only automatic **DateTime** property named **TimeStamp**.
- Add a constructor to the **HeartBeatEventArgs** class. The constructor should accept no arguments, and initialize the **TimeStamp** property to the date and time when the class is constructed. The constructor should also extend the base class constructor.

- Task 3: Declare a new delegate type

Below the **HeartBeatEventArgs** class, declare a **public delegate** type named **HeartBeatEventHandler**. The delegate should refer to a method that does not return a value, but that has the following parameters:

An **object** parameter named *sender*.

A **HeartBeatEventArgs** parameter named *args*.

- Task 4: Update the IEventEnabledMeasuringDevice interface

In the task list, locate the **TODO - Define the new event in the interface** task and then double-click this task. This task is located in the **IEventEnabledMeasuringDevice** interface

Remove this comment and add an event called **HeartBeat** to the interface. The event should specify that subscribers use the **HeartBeatEventHandler** delegate type to specify the method to run when the event is raised.

Remove the **TODO - Define the HeartBeatInterval member in the interface** comment, and then add a read-only integer property called **HeartBeatInterval** to the interface.

- Task 5: Add the HeartBeat event and HeartBeatInterval property to the MeasureDataDevice class

1. In the task list, locate the **TODO - Add the HeartBeatInterval property** task, and then double-click this task. This task is located in the **MeasureDataDevice** class.
2. Remove the **TODO - Add the HeartBeatInterval property** comment, and add a protected integer member named **heartBeatIntervalTime**.
3. Add code to implement the public integer property **HeartBeatInterval** that the **IEventEnabledMeasuringDevice** interface defines. The property should return the value of the **heartBeatInterval** member when the **get** accessor method is called. The property should have a **private set** accessor method to enable the constructor to set the property.
4. Remove the **TODO - Add the HeartBeat event** comment, and add the **HeartBeat** event that the **IEventEnabledMeasuringDevice** interface defines.
5. Remove the **TODO - add the OnHeartBeat method to fire the event** comment, and add a protected virtual void method named **OnHeartBeat** that takes no parameters.
6. In the **OnHeartBeat** method, add code to perform the following actions:

a. Check whether the **HeartBeat** event has any subscribers.

b. If the event has subscribers, raise the event, passing the current object and a new instance of the **HeartBeatEventArgs** object as parameters.

- Task 6: Use a BackgroundWorker object to generate the heartbeat

1. Remove the **TODO - Declare the BackgroundWorker to generate the heartbeat** comment, and then define a private **BackgroundWorker** object named **heartBeatTimer**.
2. Remove the **TODO - Create a method to configure the BackgroundWorker using Lambda Expressions** comment, and declare a private method named **StartHeartBeat** that accepts no parameters and does not return a value.
3. In the **StartHeartBeat** method, add code to perform the following actions:

a. Instantiate the **heartBeatTimer BackgroundWorker** object.

b. Configure the **heartBeatTimer** object to support cancellation.

c. Configure the **heartBeatTimer** object to support progress notification.

4. Add a handler for the **heartBeatTimer DoWork** event by using a lambda expression to define the actions to be performed. The lambda expression should take two parameters (use the names *o* and *args*). In the lambda expression body, add a **while** loop that continually iterates and contains code to perform the following actions:

a. Use the static **Thread.Sleep** method to put the current thread to sleep for the length of time that the **HeartBeatInterval** property indicates.

b. Check the value of the **disposed** property. If the value is **true**, terminate the loop.

c. Call the **heartBeatTimer**.**ReportProgress** method, passing zero as the parameter.

Note: Use the **+=** compound assignment operator to specify that the method will handle the **DoWork** event, define the signature of the lambda expression, and then use the **=>** operator to denote the start of the body of the lambda expression.

- Add a handler for the **heartBeatTimer**.**ReportProgress** event by using another lambda expression to create the method body. In the lambda expression body, add code to call the **OnHeartBeat** method, which raises the **HeartBeat** event.

6. At the end of the **StartHeartBeat** method, add a line of code to start the **heartBeatTimer BackgroundWorker** object running asynchronously.

- Task 7: Call the StartHeartBeat method when the MeasureDataDevice object starts running

1. In the task list, locate the **TODO - Call StartHeartBeat() from StartCollecting method** task, and then double-click this task. This task is located in the **StartCollecting** method.
2. Remove this comment, and add a line of code to invoke the **StartHeartBeat** method.

- Task 8: Dispose of the heartBeatTimer BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - dispose of the heartBeatTimer BackgroundWorker** task, and then double-click this task. This task is located in the **Dispose** method.
2. Remove the comment and add code to check that the **heartBeatTimer BackgroundWorker** object is not null. If the **heartBeatTimer** object is not null, call the **Dispose** method of the **BackgroundWorker** object.

You have now updated the **MeasureDataDevice** abstract class to implement event handlers by using lambda expressions. To enable the application to benefit from these changes, you must modify the **MeasureMassDevice** class, which extends the **MeasureDataDevice** class.

- Task 9: Update the constructor for the MeasureMassDevice class

1. Open the MeasureMassDevice class file.
2. At the start of the class, modify the signature of the constructor to take an additional **integer** value named **heartBeatInterval**.
3. Modify the body of the constructor to store the value of the **HeartBeatInterval** member in the **heartBeatInterval** member.
4. Below the existing constructor, remove the **TODO – Add a chained constructor that calls the previous constructor** comment, and add a second constructor that accepts the following parameters:
   a. A **Units** instance named **deviceUnits**.
   b. A **string** instance named **logFileName**.
5. Modify the new constructor to implicitly call the existing constructor. Pass a value of **1000** as the *heartBeatInterval* parameter value.

- Task 10: Handle the HeartBeat event in the UI

1. In the task list, locate the **TODO - Use a lambda expression to handle the HeartBeat event in the UI** task, and then double-click this task. This task is located in the **startCollecting_Click** method in the code behind the MainWindow window in the Monitor project.
2. Remove the comment, and add a lambda expression to handle the **device**.**HeartBeat** event. The lambda expression should take two parameters (name them *o* and *args*). In the body of the lambda expression, add code to update the **heartBeatTimeStamp** label with the text "HeartBeat Timestamp: *timestamp*" where timestamp is the value of the **args.TimeStamp** property.

Hint: Set the **Content** property of a label to modify the text that the label displays.

- Task 11: Test the solution

1. Build the project and correct any errors.
2. Start the application.
3. Click **Start Collecting**, and verify that values begin to appear as before. Also note that the **HeartBeat Timestamp** value now updates once per second.
4. Click **Stop Collecting**, and verify that the **RawData** list box no longer updates. Note that the timestamp continues to update, because your code does not terminate the timestamp heartbeat when you stop collecting.
5. Click **Dispose Object**, and verify that the timestamp no longer updates.

6. Close the application, and then return to Visual Studio.
7. Close Visual Studio.

Задание 3. Создание многопоточных приложений и асинхронных методов.

1. Написать приложение с визуализацией движения каких-то объектов (шариков, машинок ...), где координаты объектов вычисляются в разных потоках с разными приоритетами. Предусмотреть анализ состояния потоков через заданные промежутки времени. Синхронизацию окончания работы потоков осуществить с помощью Joun и Barrier.

2. Реализовать блокировку потока с помощью Lock и методов класса Monitor.

3. Рассмотреть реализацию мьютекса и семафора.

4.Реализовать многопоточность с помощью класса Task. В отчете описать преимущества использования новой парадигмы многопоточности в C# с использованием библиотеки TPL.

5. Реализовать многопоточный цикл с помощью Parallel.For

6. Реализовать асинхронную реализацию метода с помощью Asinc, Await.

Практическая работа №7

Тема: Практическое программирование. Работа с файлами. Паттерны проектирования ПО на C#

Цель работы: Изучить приемы практического программирования с использованием файлов. Изучить основные паттерны проектирования по книге Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона, Джона Влиссидеса ("Банда четырёх"). "Приемы объектно-ориентированного проектирования. Паттерны проектирования", разобраться с вопросом целесообразности использования паттернов в крупных проектах, научиться реализации паттернов на языке программирования C#.

Задание 1. Реализовать приложение, использующее классы File, FileInfo и FileStream, StreamReader и StreamWriter, BinaryWriter и BinaryReader.

Задание 2. Написать в отчет о практической работе небольшой авторский реферат о паттернах проектирования:

Описать структуры паттернов:

1. Порождающие паттерны Порождающие паттерны — это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов. Среди них выделяются следующие: Абстрактная фабрика (Abstract Factory) Строитель (Builder) Фабричный метод (Factory Method) Прототип (Prototype) Одиночка (Singleton)
2. Структурные паттерны - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся: Адаптер (Adapter) Мост (Bridge) Компоновщик (Composite) Декоратор (Decorator) Фасад (Facade) Приспособленец (Flyweight) Заместитель (Proxy)
3. Поведенческие паттерны - определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие: Цепочка обязанностей (Chain of responsibility) Команда (Command) Интерпретатор (Interpreter) Итератор (Iterator) Посредник (Mediator) Хранитель (Memento) Наблюдатель (Observer) Состояние (State) Стратегия (Strategy) Шаблонный метод (Template method) Посетитель (Visitor).
   Включить в реферат различные точки зрения на использование паттернов, изученные вами на специализированных сайтах и форумах.

Задание 3. Реализовать 6 паттернов проектирования не менее 3 типов на языке программирования C#.

В качестве основы для проекта можно выбрать раздел Интернет-магазина, торгующего различными товарами (добиться расширяемости и масштабируемости проекта).
Можно выбрать сложную иерархитческую структуру объектов и классов, типа студент, преподаватель, менеджер, руководитель и т.п.

Практическая работа №8

Тема: Коллекции, индексаторы, словари, LINQ, сериализация JSON

Цель работы: Научиться на практике использовать списки, «резиновые» массивы, обобщенные коллекции, словари, индексаторы и «умные» массивы. Разобраться с технологией LINQ и научиться сериализовывать и десериализовывать объекты.

Ключевые понятия: Array, ArrayList, foreach, Collections, IEnumerable и IEnumerator, Yield, ICollection, IDictionary, Queue, Stack, BitArray, Indexer .

Перед выполнением лабораторной работы рекомендуем:

1. Изучить презентацию лектора курса: «Коллекции» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка.

Содержание отчета:

4. Описание и раскрытие ключевых понятий
5. Краткое описание изученной литературы
6. Программный код для каждого из заданий с подробными комментариями

Задание 1. Коллекции

1. Реализовать коллекцию, состоящую из разнородных элементов. Применить к коллекции методы: Add, CopyTo, Clear, IndexOff, Insert, Remove, Reverse, Sort.
2. Написать свой пример реализации интерфейсов IEnumerable и IEnumerator.
3. Написать пример реализации с помощью коллекции стека, или очереди. В отчете объяснить преимущество этой реализации перед использованием массивов.

Задание 2. Словари

1. Реализовать свой собственный словарь Dictionery. Измерить скорость выборки значений по ключу.
2. Реализовать с помощью индексатора «умный массив», который не позволяет выйти за границы массива и использовать в качестве значений отрицательные числа.

Задание 3. LINQ

1. Реализовать свой пример использования операторов запросов и методов расширений LINQ.

Задание 4. Сериализация

1. Реализовать сериализацию и десериализацию экземпляра класса в JSON

Основная учебная литература:

1. Прайс Марк. C# 10 и .NET 6. Современная кросс-платформенная разработка. — СПб.: Питер, 2023. — 848 с.: ил. — (Серия «Для профессионалов»).
2. Залогова, Л.А. Основы объектно-ориентированного программирования на базе языка C# : учебное пособие / Л.А. Залогова. — 2-е изд., стер. — Санкт-Петербург : Лань, 2020. — 192 с. — ISBN 978-5-8114-4757-2. — Текст : электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/126160 (дата обращения: 31.10.2019). — Режим доступа: для авториз. пользователей.
3. Барков, И.А. Объектно-ориентированное программирование : учебник / И.А. Барков. — Санкт-Петербург : Лань, 2019. — 700 с. — ISBN 978-5-8114-3586-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/119661 (дата обращения: 18.01.2020). — Режим доступа: для авториз. пользователей.

Дополнительная учебная литература:

1. Корпорация Microsoft. C#. Спецификация языка. Версия 5.0. (входит в комплект поставки Visual Studio и находится в каталоге Program Files\Microsoft Visual Studio. Венц К.
2. Евдокимов П. В., Дубовик Е. В. Справочник C#. Кратко, быстро, под рукой — СПб.: Издательство Наука и Техника, 2023. — 336 с., ил.
3. Безопасность ASP.NET Core / пер. с англ. Д. А. Беликова. - М.: ДМК Пресс, 2023. - 386 с.: ил.
4. Шилдт Герберт. Полный справочник по C#. Пер. с англ. - М.: Издательский дом "Вильямс", 2008. - 752с.: ил.
5. Ватсон Б.C# 4.0 на примерах. — СПб.: БХ В-Петербург, 2011. — 608 с.: ил.
6. Зиборов В. В.Visual C# 2012 на примерах. — СПб.: БХВ-Петербург, 2013. — 480 с.: ил.
7. Климов А. П.C#. Советы программистам. — СПб.: БХВ-Петербург, 2008. 544 с.: ил.
8. Хейлсберг А. Торгерсен М., Вилтамут С., Голд П. Язык программирования C#. Классика Computers Science. 4-е изд. — СПб.: Питер, 2012. — 784 с.: ил.
9. Жарков В.А. Компьютерная графика, мультимедиа и игры на Visual C# 2005. – М.: Жарков Пресс, 2005. – 812 с.
10. Нэш, Трей. C# 2010: ускоренный курс для профессионалов. : Пер. с англ. — М. : ООО “И.Д.Вильямс”, 2010. — 592 с.
11. Троелсен. Эндрю.Язык программирования C# 2010 и платформа .NET 4.0. 5-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 1392 с. : ил. — Парал. тит. англ.
12. Рихтер Дж.CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е изд. — СПб.: Питер, 2012. — 928 с.: ил.

Интернет ресурсы:

1. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
2. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info