Объектно-ориентированное программирование на C#

Лабораторные работы

Лабораторная работа 1. Знакомство с языком C#

Цель – получение опыта разработки простейших приложений на языке C#.

Теоретическая справка

C# — объектно-ориентированный язык программирования общего назначения. Разработан в 1998-2001 годах группой инженеров компании Microsoft. C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов, делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML. (Источник: Википедия)

Рассмотрим пример, демонстрирующий консольный ввод и вывод.

Листинг 1. Консольный ввод и вывод

```
static void Main(string[] args)
{
    Console.WriteLine("Введите Ваше имя, пожалуйста: ");
    /* Вопросительный знак указывает,
     * что переменная может хранить значение null: */
    string? UserName = Console.ReadLine();
    // Один способ вывода переменной:
    Console.WriteLine("Здравствуйте, " + UserName);
    // Другой способ вывода переменной:
    Console.WriteLine($"Сколько Вам лет, {UserName}?");
    int Age = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine($"Ого! Вам уже {Age}");
}
```

C#, как упоминалось ранее, является *полноценным объектно-ориентированным языком*. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов. Описанием объекта является **класс**, а объект представляет **экземпляр этого класса**. Класс определяется с помощью ключевого слова `class`. Рассмотрим пример простейшего класса с одним полем, одним конструктором и одним методом.

Листинг 2. Простейший класс

```
class Person
{
    public string? Name;
    public Person(string Name) { this.Name = Name; }
    public void SayHi() { Console.WriteLine("Привет, меня зовут " + Name); }
}
```

Создадим экземпляр данного класса и воспользуемся методом `SayHi()`:

Листинг 3. Использование класса Person

```
static void Main(string[] args)
{
    Person Vitalya = new("Виталя Волков");
    Vitalya.SayHi();
}
```

Прим. Подробнее ознакомиться с членами класса можно по ссылке: https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/members

---

Полезно

Пара рекомендаций о названиях классов, полей и т.п. от Microsoft:

- «Используйте регистр pascal ("PascalCasing") при именовании классов (`class`), записей (`record`), структур (`struct`) и публичных (`public`) членов класса»

Листинг 4. Пример использования PascalCasing

```
public class DataService
{
}
```

- «Используйте "camelCasing" при именовании `private` или `internal` полей, а также добавляйте префикс _»

Листинг 5. Пример использования camelCasing

```
class Person
{
    private int _age;
```

Задание

Создать приложение, удовлетворяющее следующим требованиям:

- Приложение должно содержать класс `Student`;
- В классе `Student` должны быть закрытое (`private`) поле _name строкового типа, публичное (`public`) поле Age целочисленного типа и методы, устанавливающие и возвращающие данные значения («геттеры» и «сеттеры»);
- Класс `Student` должен иметь конструктор, принимающий в качестве аргумента только имя, и конструктор, принимающий два аргумента;
- Класс `Student` должен иметь публичный метод – функцию `WriteInfo()`, которая возвращает информацию о студенте в виде строки;

- Класс `Student` должен иметь публичный метод – процедуру `BecomeOlder()`, увеличивающую возраст студента на единицу.

В приложении продемонстрировать использование обоих конструкторов и всех методов класса.

Задание на доп. баллы

- Реализовать «геттеры» и «сеттеры» при помощи свойств, о которых доп. информацию можно получить по ссылке: https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/properties.

Лабораторная работа 2. Модификаторы

Цель – закрепление знаний о модификаторах доступа и модификаторах `const`, `ref`, `in` и `out`.

Теоретическая справка

Для объявления константного поля или локальной постоянной используется ключевое слово `const`. Константные поля и локальные постоянные не являются переменными и не могут быть изменены.

Все поля, методы и остальные компоненты класса имеют **модификаторы доступа**. Модификаторы доступа *позволяют задать допустимую область видимости для компонентов класса*. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В языке C# применяются следующие модификаторы доступа:

- `private`: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
- `private protected`: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке (подробнее о наследовании и производных классах в одной из следующих работ).
- `file`: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с таким модификатором доступны только из текущего файла кода.
- `protected`: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
- `internal`: компонент класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
- `protected internal`: совмещает функционал двух модификаторов `protected` и `internal`. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.

- `public`: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

Существует два способа передачи параметров в метод в языке C#: **по значению** и **по ссылке**.

Наиболее простой способ передачи параметров – передача по значению, это обычный способ передачи параметров.

Листинг 6. Передача параметра по значению

```
static void Main(string[] args)
{
    int a = 5;
    Console.WriteLine("До: " + a);
    Increase(a);
    Console.WriteLine("После: " + a);
}

static void Increase(int a)
{
    a++;
}
```

```
До: 5
После: 5
```

Рисунок 1. Передача параметра по значению

При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее *работает с этой копией независимо от самой переменной*.

При передаче параметров по ссылке перед параметрами используется модификатор `ref`.

Листинг 7. Передача параметров по ссылке

```
static void Main(string[] args)
{
    int a = 5;
    Console.WriteLine("До: " + a);
    Increase(ref a);
    Console.WriteLine("После: " + a);
}

static void Increase(ref int a)
{
    a++;
}
```

```
До: 5
После: 6
```

Рисунок 2. Передача параметров по ссылке

В C# можно сделать параметр выходным, если поставить перед ним модификатор `out`. Это позволяет вернуть из метода не одно значение, а несколько.

Кроме выходных параметров с модификатором `out`, метод может использовать входные параметры с модификатором `in`. Модификатор `in` указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить.

Листинг 8. Использование модификаторов in и out

```
static void Main(string[] args)
{
    int a = 5;
    Console.WriteLine("Сторона: " + a);
    int area, perimeter;
    GetAreaAndPerimeter(in a, out area, out perimeter);
    Console.WriteLine("Площадь: " + area);
    Console.WriteLine("Периметр: " + perimeter);
}

static void GetAreaAndPerimeter(in int a, out int area,
    out int perimeter)
{
    // a = 10; // Ошибка: a доступна только для чтения
    area = a * a;
    perimeter = a * 4;
}
```

```
Сторона: 5
Площадь: 25
Периметр: 20
```

Рисунок 3. Использование модификаторов in и out

Задание

- Добавить в программу, разработанную в ходе первой работы, два класса: `Subject` и `Game`. Добавить объекты данных классов в качестве полей, отражающих любимые предмет и игру студента;
- Продемонстрировать отличие поля с модификатором `private` от поля с модификатором `public`;
- Продемонстрировать отличие передачи параметра по значению от передачи по ссылке на примере объекта класса `Student`;
- Написать метод, возвращающий любимые предмет и игру студента.

Лабораторная работа 3. Классы

Цель – закрепление знаний о конструкторах, инициализаторах, деструкторах, свойствах и модификаторе `static`.

Теоретическая справка

Каждый раз, когда создается класс или структура, вызывается **конструктор**. Класс или структура может иметь несколько конструкторов, принимающих различные аргументы. Конструкторы позволяют программисту задавать значения по умолчанию и писать гибкий и удобный для чтения код.

Например, данный класс `Person` имеет 2 конструктора:

Листинг 9. Класс Person

```
class Person
{
    public string Name;
    public int Age;
    public Person()
    {
        Name = "Tom";
        Age = 37;
    }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public void Print()
    {
        Console.WriteLine($"Имя: {Name}  Возраст: {Age}");
    }
}
```

Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта.

Листинг 10. Применение конструкторов и инициализаторов

```
static void Main(string[] args)
{
    // Используется первый конструктор
    Person tom = new();
    tom.Print();
    // Используется второй конструктор
    Person billy = new("Billy", 56);
    billy.Print();
    // Используется инициализатор
    Person mark = new() { Name = "Mark", Age = 46 };
    mark.Print();
}
```

В языке C# неиспользуемые объекты автоматически удаляются *сборщиком мусора*. **Методы завершения** (также называемые деструкторами) используются для любой необходимой окончательной очистки, когда сборщик мусора окончательно удаляет экземпляр класса.

Листинг 11. Метод завершения

```
~Person()
{
    Console.WriteLine($"Прощай, {Name}");
}
```

Методы завершения невозможно вызвать. Они запускаются автоматически. Метод завершения не принимает модификаторов и не имеет параметров.

Свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения закрытого (`private`) поля. Свойства можно использовать так, как если бы они являлись публичными полями. Эта функция позволяет легко получать доступ к данным и по-прежнему способствует обеспечению безопасности и гибкости методов.

Стандартное описание свойства имеет следующий синтаксис:

Листинг 12. Синтаксис описания свойства

```
[модификаторы] тип_свойства название_свойства;
{
    get { действия, выполняемые при получении значения свойства}
    set { действия, выполняемые при установке значения свойства}
}
```

Листинг 13. Пример использования свойства

```
public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name {
        get => _name;
        set => _name = value;
    }

    public decimal Price {
        get => _cost;
        set => _cost = value;
    }
}
```

Аналогичная запись:

Листинг 14. Альтернативный вариант описания свойства

```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

Кроме обычных полей, методов, свойств *классы и структуры могут иметь статические поля, методы, свойства*. Статические поля, методы, свойства относятся ко всему классу/всей структуре, и для обращения к подобным членам необязательно создавать экземпляр класса / структуры.

Статическое поле определяется так же, как и обычное, только перед типом поля указывается ключевое слово `static`. Подобным образом мы можем создавать и использовать статические свойства.

Листинг 15. Создание статических членов класса

```
public class SaleItem
{

    public string Name
    { get; set; }

    public static decimal Price
    { get; set; }

    public static void WriteInfo()
    {
        Console.WriteLine($"Товары стоят {Price} форинтов");
    }
}
```

Листинг 16. Обращение к статическому свойству Price.

```
static void Main(string[] args)
{
    SaleItem good = new();
    good.Name = "Картошка";
    SaleItem.Price = 100;
    Console.WriteLine($"{good.Name} стоит {SaleItem.Price} форинтов");

    SaleItem good2 = new();
    good2.Name = "Морковка";
    Console.WriteLine($"{good2.Name} тоже стоит {SaleItem.Price} форинтов.");
    Console.WriteLine($"Всё стоит {SaleItem.Price} форинтов.");
}
```

Статические методы определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам точно так же применяется имя класса / структуры.

Листинг 17. Использование статического метода

```
static void Main(string[] args)
{
    SaleItem.Price = 150;
    SaleItem.WriteInfo();
    Console.WriteLine("Инфляция!");
}
```

```
Товары стоят 150 форинтов
Инфляция!
```

Рисунок 4. Использование статического метода

Кроме обычных конструкторов, у класса также могут быть **статические конструкторы**. Статические конструкторы имеют следующие отличительные черты:

- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров.
- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово `this` для ссылки на текущий объект класса. Можно обращаться только к статическим членам класса.
- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам.
- Статические конструкторы обычно используются для инициализации статических данных либо выполняют действия, которые требуется выполнить только один раз.

Статические классы объявляются с модификатором `static` и могут содержать только статические поля, свойства и методы.

Прим. Подробнее о свойствах можно прочесть: https://goo.su/voshUa

---

Задание

- Взять в качестве основы программу, разработанную в ходе первой работы;
- Создать три объекта класса `Student` различными способами;
- Добавить в класс `Student` статическое поле, статическое свойство, статический метод и статический конструктор;
- Разработать статический класс и связать его с классом `Student`.

Лабораторная работа 4. Наследование

Цель – изучить реализацию наследования в языке C#.

Теоретическая справка

Наследование, вместе с **инкапсуляцией** и **полиморфизмом**, является одной из трех основных характеристик объектно-ориентированного программирования. *Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах*.

Класс, члены которого наследуются, называется **базовым классом**, а класс, который наследует эти члены, называется **производным классом**. Производный класс может иметь только один прямой базовый класс. Однако наследование является транзитивным. Если `ClassC` является производным от `ClassB`, а `ClassB` — от `ClassA`, `ClassC` наследует члены, объявленные в `ClassB` и `ClassA`.

Пример наследования: класс `Mouse` является производным от класса `Animal`.

Листинг 18. Пример наследования

```
class Animal
{
    public string Name;
    public Animal()
    {
        Name = "Джерри";
    }

    public void PrintName()
    {
        Console.WriteLine($"Имя: {Name}");
    }
}

class Mouse : Animal
{
    public int TailLength;
    public Mouse()
    {
        TailLength = 5;
    }
}
```

Листинг 19. Пример использования метода базового класса объектом производного класса

```
static void Main(string[] args)
{
    Mouse mouse = new();
    mouse.PrintName();
}
```

По умолчанию все классы наследуются от базового класса `Object`, даже если мы явным образом не устанавливаем наследование. Поэтому упомянутые классы кроме своих собственных методов также будут иметь и методы класса `Object`: `ToString()`, `Equals()`, `GetHashCode()` и `GetType()`.

С помощью ключевого слова `base` можно обратиться к базовому классу.

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае *класс-наследник может переопределять методы и свойства базового класса*.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором `virtual`. Такие методы и свойства называют виртуальными.

Чтобы **переопределить метод** в классе-наследнике, этот метод определяется с модификатором `override`. Переопределенный метод в классе-наследнике *должен иметь тот же набор параметров, что и виртуальный метод в базовом классе*.

Листинг 20. Переопределение метода

```
class Animal
{
    public string Name;
    public Animal() { Name = "Джерри"; }

    public virtual void PrintName()
    {
        Console.WriteLine($"Имя: {Name}");
    }
}

class Mouse : Animal
{
    public int TailLength;

    public Mouse() { TailLength = 5; }

    public override void PrintName()
    {
        Console.WriteLine($"У мышки тоже есть имя: {base.Name}");
    }
}
```

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором `sealed`.

Другим способом изменить функциональность метода, унаследованного от базового класса, является **скрытие**. Скрытие метода/свойства представляет

определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для скрытия членов класса применяется ключевое слово `new`.

Листинг 21. Скрытие

```
class Animal
{
    public string Name;
    public Animal()
    {
        Name = "Джерри";
    }

    public void PrintName()
    {
        Console.WriteLine($"Имя: {Name}");
    }
}

class Mouse : Animal
{
    public int TailLength;

    public Mouse()
    {
        TailLength = 5;
    }

    public new void PrintName()
    {
        Console.WriteLine($"У мышки тоже есть имя: {base.Name}");
    }
}
```

Кроме обычных классов в C# есть **абстрактные классы**. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. При определении абстрактных классов используется ключевое слово `abstract`. Главное отличие абстрактных классов от обычных состоит в том, что мы не можем использовать конструктор абстрактного класса для создания экземпляра класса. Тем не менее абстрактные классы полезны для описания некоторого общего функционала, который могут наследовать и использовать производные классы.

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова `abstract` и не имеют никакого функционала. *Производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе*.

Задание

- Взять в качестве основы программу, разработанную в ходе первой работы;
- Создать для класса `Student` базовый *абстрактный* класс `Person`, имеющий хотя бы одно абстрактное поле и один абстрактный метод;
- Разработать для класса `Student` *производный* класс `ITStudent`;
- Продемонстрировать на примере классов `Student` и `ITStudent` различие между переопределением и скрытием метода;
- Переопределить для класса `Student` метод `ToString()` класса `Object`.

Лабораторная работа 5. Интерфейсы

Цель – изучить применение интерфейсов в языке C#.

Теоретическая справка

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово `interface`. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, `IComparable`, `IEnumerable`.

Нельзя создавать объекты интерфейса напрямую с помощью конструктора.

При применении интерфейса, как и при наследовании, после имени класса или структуры указывается двоеточие, затем идут названия применяемых интерфейсов. *Класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию*.

Листинг 22. Применение интерфейса

```
interface ICreature
{
    void Eat();
}

class Animal : ICreature
{
    public string Name;

    public Animal()
    {
        Name = "Джерри";
    }

    public void Eat()
    {
        Console.WriteLine("Я ем. Кого-то или что-то");
    }
}
```

В C# классы и структуры могут реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую.

Кроме рассмотренного неявного применения интерфейсов, существует также **явная реализация интерфейса**. При явной реализации указывается название метода или свойства вместе с названием интерфейса, при этом мы не можем использовать модификатор `public`, то есть методы являются закрытыми.

Листинг 23. Явная реализация интерфейса

```
interface ICreature
{
    void Eat();
}

class Animal : ICreature
{
    public string Name;
    public Animal()
    {
        Name = "Джерри";
    }

    void ICreature.Eat()
    {
        Console.WriteLine("Я ем. Кого-то или что-то");
    }
}
```

При явной реализации интерфейса его методы и свойства не являются частью интерфейса класса. Поэтому напрямую через объект класса мы к ним обратиться не сможем.

Явная реализация может понадобиться, например, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров.

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса, которые не имеют реализации по умолчанию. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы.

Интерфейсы, как и классы, могут наследоваться.

Листинг 24. Применение нескольких интерфейсов

```
interface IPrey
{
    void Eat();
}

interface IPredator
{
    void Eat();
}

class Animal : IPrey, IPredator
{
    public string Name;
    public Animal() { Name = "Джерри"; }

    void IPredator.Eat() { Console.WriteLine("Я кого-то ем"); }

    void IPrey.Eat() { Console.WriteLine("Меня кто-то ест!"); }
}
```

Задание

- Взять в качестве основы программу, разработанную в ходе четвёртой работы;
- Абстрактный класс `Person` заменить интерфейсом `IPerson`. Самостоятельно определить функциональность данного интерфейса;
- Для класса `ITStudent` разработать интерфейс `ISpecialist`, производный от `IPerson`. Продемонстрировать явную реализацию интерфейса. Самостоятельно определить функциональность данного интерфейса;
- Разработать класс `Subject`. Добавить в класс `Student` объект этого класса в качестве поля, отражающего любимый предмет студента;
- Реализовать для класса `Student` встроенные интерфейсы `ICloneable` и `IComparable`.

Лабораторная работа 6. Обработка исключений

Цель – знакомство обучающихся с обработкой исключений в C#.

Теоретическая справка

В ходе выполнения предыдущих заданий вы наверняка хотя бы один раз столкнулись с неожиданной ошибкой. Для обработки таких ситуаций в C# предназначена конструкция `try...catch...finally`.

Логика их выполнения будет следующая: сначала выполняются все инструкции в блоке `try`. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок `finally`. И затем конструкция завершает свою работу.

При возникновении исключения CLR (общеязыковая исполняющая среда) разворачивает стек в поисках метода с блоком перехвата для конкретного типа исключения и выполняет первый найденный блок перехвата. Во многих случаях исключение может быть вызвано не тем методом, который был вызван непосредственно вашим кодом, а *другим методом, расположенным дальше в стеке вызовов*. Если нигде в стеке вызовов не найдется подходящего блока перехвата, он завершит процесс и выдаст пользователю сообщение об ошибке. Если нужный блок `catch` найден, то он выполняется, и после его завершения выполняется блок `finally`.

Базовым для всех типов исключений является тип `Exception`. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

- `InnerException`: хранит информацию об исключении, которое послужило причиной текущего исключения;
- `Message`: хранит сообщение об исключении, текст ошибки;
- `Source`: хранит имя объекта или сборки, которое вызвало исключение;
- `StackTrace`: возвращает строковое представление стека вызовов, которые привели к возникновению исключения;
- `TargetSite`: возвращает метод, в котором и было вызвано исключение.

Так как тип `Exception` является базовым типом для всех исключений, выражение `catch (Exception ex)` будет обрабатывать все исключения, которые могут возникнуть. Однако, есть более специализированные типы исключений,

которые предназначены для обработки каких-то определенных видов исключений. Их иерархию можно представить следующим образом:

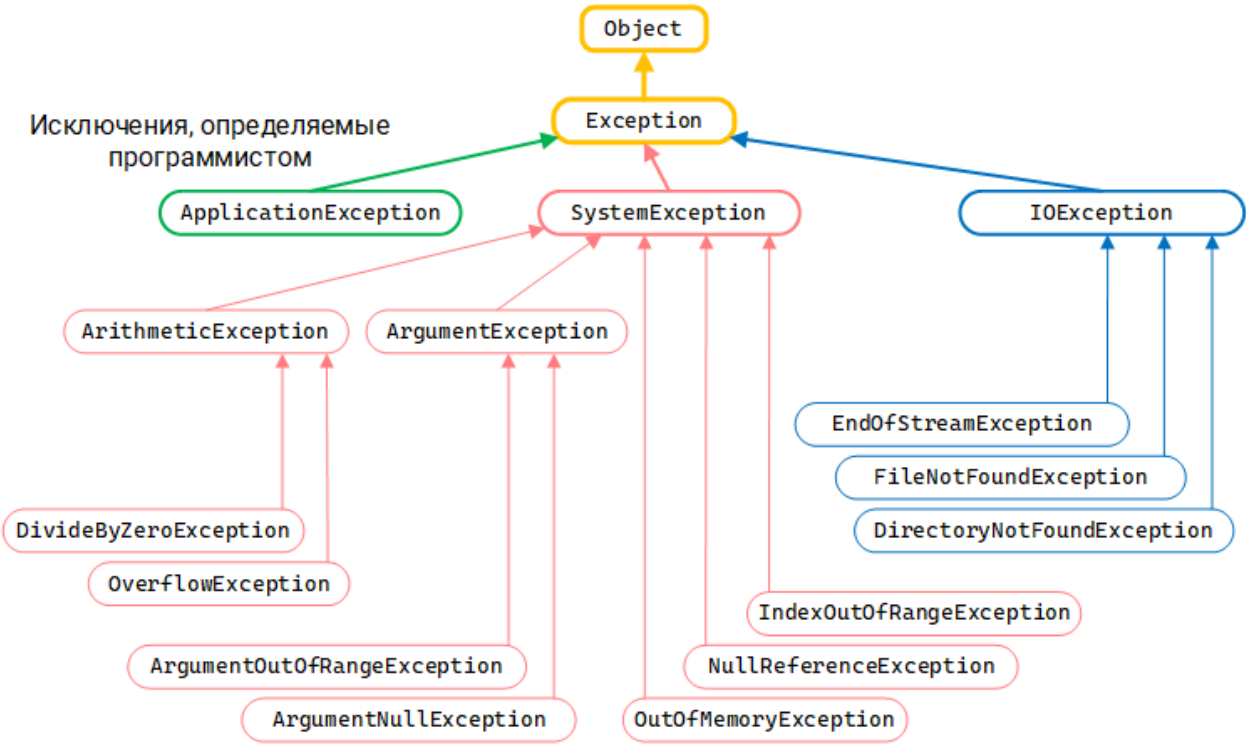

Рисунок 5. Иерархия исключений в C# [1]

Приведем пример, в котором можно проследить поиск блока перехвата и обработку определенных видов исключений. Для этого обратимся к частой ошибке, которая закрадывается в код из-за невнимательности программиста: выходу за пределы массива.

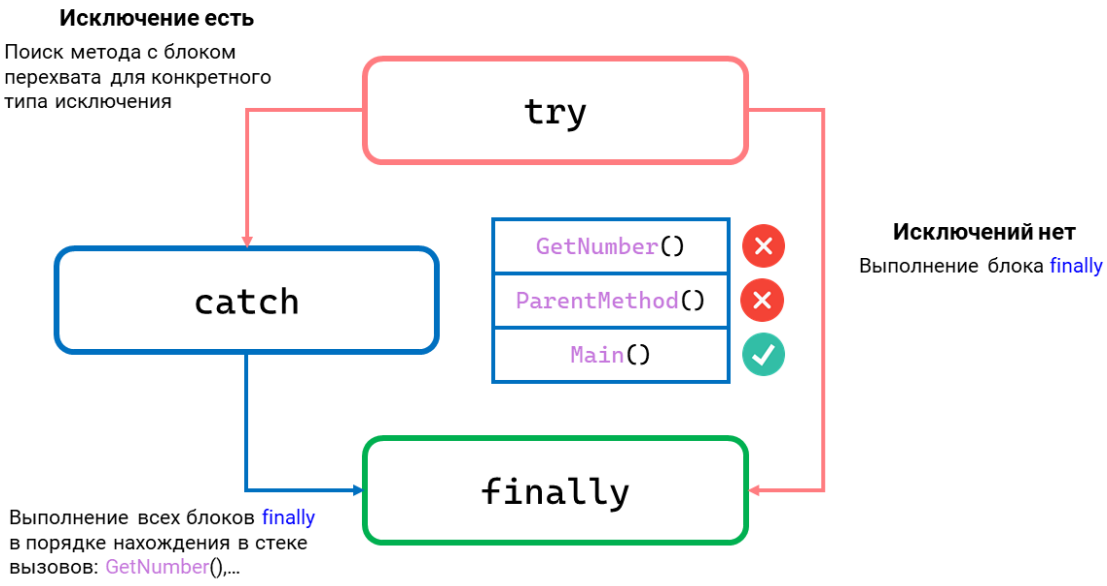

Рисунок 6. Логика работы конструкции try…catch..finally в примере

Листинг 25. Пример обработки исключений

```
public class ExceptionTest
{
    public static void Main()
    {
        DemoException ex = new DemoException();

        try
        {
            Console.WriteLine($"First test: {ex.ParentMethod(5)}, Second test:
{ex.ParentMethod(7)}");
        } catch (IndexOutOfRangeException e) {
            Console.WriteLine("Out of range.\n" +
                "Message: " + e.Message + "\n Stack trace: " + e.StackTrace);
        } finally {
            Console.WriteLine("Main finally");
        }
    }
}

public class DemoException
{
    public int ParentMethod(int numberPosition)
    {
        int number = 0;
        number = GetNumber(numberPosition);
        return number;
    }

    public int GetNumber(int numberPosition)
    {
        int number = 0;
        int[] a = { 0, 1, 2, 3, 4, 5 };
        try
        {
            number = a[numberPosition];
        } catch (ArgumentOutOfRangeException ex) when (numberPosition < 0) {
            Console.WriteLine("Position can't be less then 0.");
        }
        finally {
            Console.WriteLine("GetNumber finally");
        }
        return number;
    }
}
```

При попытке запустить эту программу мы получим следующее сообщение:

```
GetNumber finally
GetNumber finally
Out of range.
Message: Index was outside the bounds of the array.
 Stack trace:    at DemoException.GetNumber(Int32 numberPosition) in G:\University\3rd course\C#\University System
\Lab 6\Lab 6\Lab 6\Program.cs:line 34
   at DemoException.ParentMethod(Int32 numberPosition) in G:\University\3rd course\C#\University System\Lab 6\Lab
6\Lab 6\Program.cs:line 24
   at ExceptionTest.Main() in G:\University\3rd course\C#\University System\Lab 6\Lab 6\Lab 6\Program.cs:line 9
Main finally
```

Рисунок 7. Результат, полученный в результате обработки исключения

Ошибка была поймана `catch`'ем, который находился на 2 уровня ниже в стеке вызовов, а `StackTrace` позволил нам проследить, в каком моменте она появилась.

В приведенном выше коде также использовался **фильтр**, который задается с помощью выражения `when`, после которого в скобках указывается условие.

До этого момента отлов всех ошибок у нас происходил в автоматическом режиме, но язык C# также позволяет генерировать исключения вручную с помощью оператора `throw`. После него указывается объект исключения, через конструктора которого мы можем передать сообщение об ошибке.

Конструкция `try…catch…finally` – очень мощный инструмент для обработки ошибок, но *пользоваться им надо с умом, потому что каждый его дополнительный вызов будет снижать производительность вашей программы*.

Задание

- Рассмотреть класс `Student` из лаб. работы №4 и продумать для него систему обработки возможных исключений.
- Применить при разработке оператор `throw`.
- Воспользоваться фильтром `when`.

Прим. В качестве материала для самостоятельного изучения почитать про лучшие методы работы с исключениями: https://goo.su/D8UI8U

Библиографический список

---

1. Mok H. N. From Java to C#: A Developer's Guide / H. N. Mok. – Great Britain : Pearson Education Ltd, 2003. – 464 с. – ISBN 0-321-13622-5

Лабораторная работа 7. Простейшее графическое приложение

Цель – формирование опыта создания простого GUI приложения с использованием Windows Forms.

Теоретическая справка

Windows Forms — это платформа, которая позволяет создавать настольные приложения с помощью графического пользовательского интерфейса. Она обеспечивает один из самых эффективных способов создания классических приложений с помощью визуального конструктора в Visual Studio. Размещение визуальных элементов управления, таких как кнопки, текстовые поля и др., прямо на холст путем перетаскивания упрощает создание классических приложений.

Ознакомимся с первым окном, которое открылось после создания проекта. Это форма, на которой будут располагаться будущие элементы нашего приложения. При нажатии на кнопку “*Toolbox*” в правой части экрана откроется перечень доступных элементов управления (альтернативный способ открытия – *Ctrl + Alt + X*).

Для того чтобы открыть код формы, достаточно нажать правой кнопкой мыши по пустому пространству и выбрать пункт “*View Code*”.

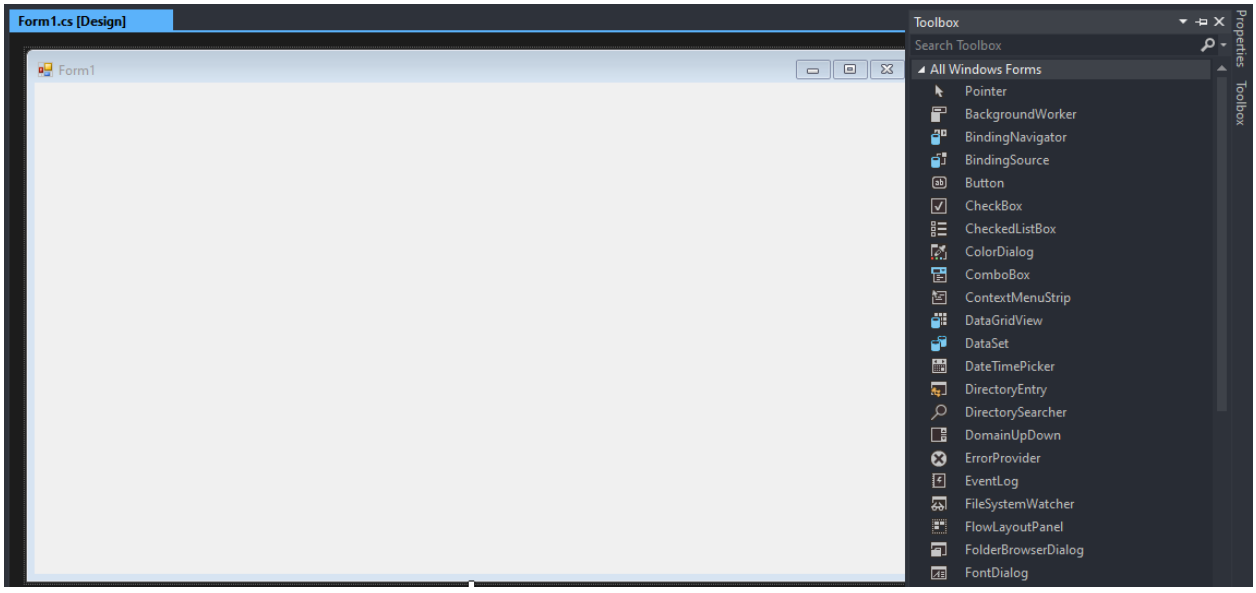

Рисунок 8. Новая форма и окно “Toolbox” с элементами управления

Добавим на форму четыре элемента: `DataGridView`, `ComboBox`, `TextBox`, `Button` и реализуем следующую логику: при нажатии на кнопку активный текст из `ComboBox` и `TextBox` будет добавляться в `DataGridView`.

Для начала создадим класс `WeekDay`, который будет отображать нашу модель данных:

Листинг 26. Реализация класса WeekDay

```
public class WeekDay
{
    private int _id;
    private string _dayName;

    public int Id { get => _id; set => _id = value; }
    public string DayName { get => _dayName; set => _dayName = value; }
}
```

Далее инициализируем `ComboBox`. Производить эту операцию мы будем при загрузке формы. Для этого откроем свойства формы (ПКМ => “*Properties*”), выберем иконку молнии в верхней части экрана (события) и дважды нажмем на пустое поле справа от надписи “*Load*”. Введем туда следующий код:

Листинг 27. Инициализация ComboBox

```
private void initWeekDays()
{
    List<WeekDay> weekDays = new List<WeekDay> {
                new WeekDay{ Id = 1, DayName = "Monday" },
                new WeekDay{ Id = 2, DayName = "Tuesday" },
                new WeekDay{ Id = 3, DayName =  "Wednesday" },
                new WeekDay{ Id = 4, DayName =  "Thursday" },
                new WeekDay{ Id = 5, DayName =  "Friday" },
                new WeekDay{ Id = 6, DayName =  "Saturday" },
                new WeekDay{ Id = 7, DayName =  "Sunday" },
            };
    comboBox1.DataSource = weekDays;
    comboBox1.ValueMember = "Id";
    comboBox1.DisplayMember = "DayName";
}

private void Form1_Load(object sender, EventArgs e)
{
    initWeekDays();
}
```

Таким образом, при загрузке форме будет вызываться процедура `initWeekDays`(), которая производит инициализацию `ComboBox` (заполняет его экземплярами класса `WeekDay`).

По аналогичному алгоритму добавим событие `Click` для нашей кнопки и добавим туда следующий код:

Листинг 28. Реализация процедуры добавления строки в DataGridView

```
private void button1_Click(object sender, EventArgs e)
{
    dataGridView1.Rows.Add(((WeekDay)comboBox1.SelectedItem).DayName.ToString(),
        textBox1.Text);
}
```

Теперь при каждом нажатии на кнопку мы будем добавлять в `DataGridView` новую строку, состоящую из дня недели и комментариев к нему.

> **Прим.** Для того чтобы изменить псевдоним элемента управления, нужно отредактировать свойство “*Name*”.

Задание

Разработать форму для визуальной работы с классом `Student`.

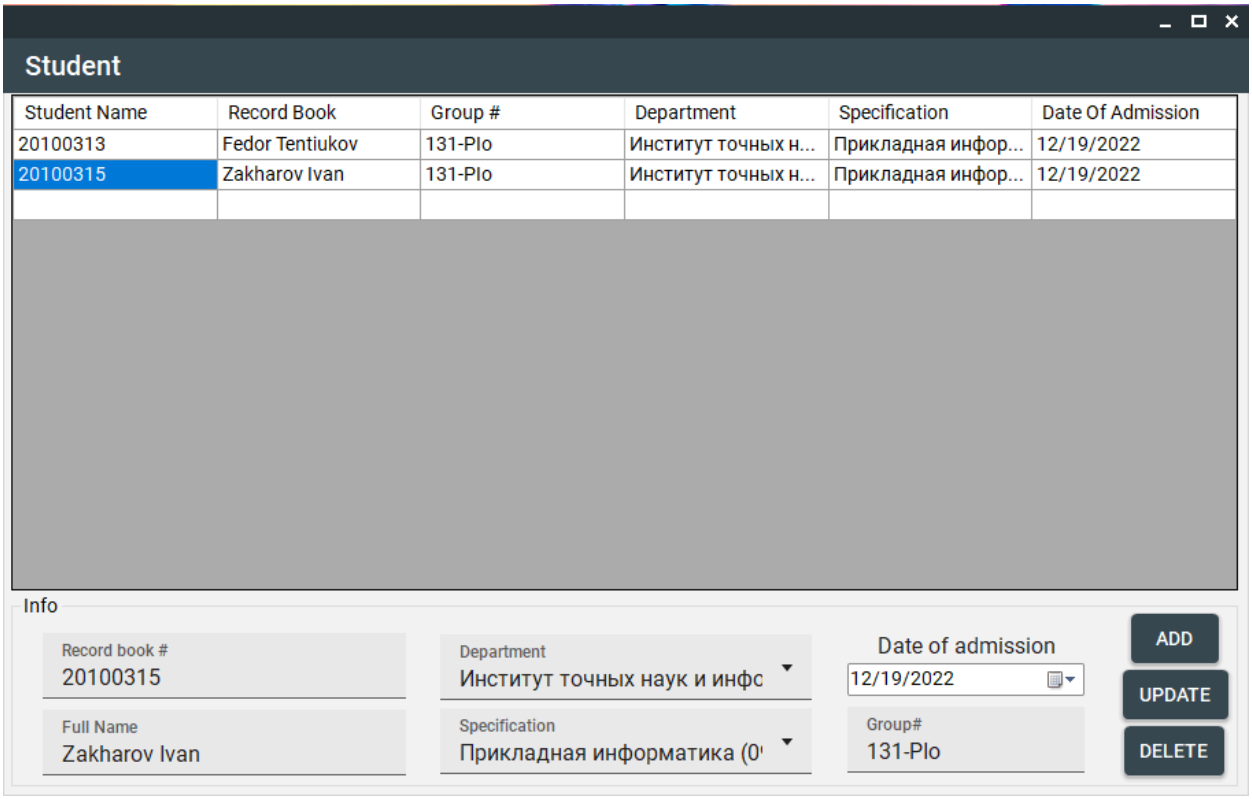

Рисунок 9. Примерный интерфейс программы для работы с классом Student

Требования:

- Класс `Student` должен содержать следующие свойства: recordBook (№ студ. билета), fullName (ФИО студента), department (институт), specification (направление), dateOfAdmission (дата зачисления), group (группа);
- Присутствие функций добавления, чтения, изменения и удаления записей;
- Валидация ввода (не может быть два студента с одинаковым номером зачетки);
- Визуализация группы с использованием элемента управления `DataGridView`;
- Динамическое обновление данных (без кнопок по типу “Update data”).

Задание на доп. баллы

- Каскадное обновление `ComboBox` (в specificationCB отображаются только те направления, которые относятся к выбранному институту);
- Адаптивный интерфейс.

Лабораторная работа 8. MVP и работа с базами данных

Цель – знакомство с MVP паттерном, формирование навыков взаимодействия с базами данных и разработки архитектуры приложений.

Теоретическая справка

Работа с базами данных – неотъемлемая часть разработки программного обеспечения. Нередко такие проекты достигают невероятных масштабов, после чего ориентироваться в таком проекте становится значительно сложней.

Для того чтобы свести такие ситуации к минимуму, были разработаны **паттерны**, которые позволяют структурировать код графического приложения определенным образом.

Сегодня речь пойдет о **MVP** (Model-View-Presenter) – наиболее подходящем паттерне для работы с Windows Forms.

> **Прим.** С другими паттернами можно ознакомиться по ссылке: https://habr.com/ru/post/215605/

Согласно MVP, приложение делится на 3 части:

Model (модель) – уровень данных. Основная задача – работа с данными и руководство всеми бизнес-процессами. *Модель должна быть полностью независима от остальных частей продукта*.

Листинг 29. Предлагаемый шаблон реализации класса StudentModel

```
public class StudentModel
{
    private int _id;
    private string _name;
    private string _recordBook;
    // More properties...

    public int Id {
        get => _id; set => _id = value;
    }
    public string Name {
        get => _name; set => _name = value;
    }
    public string RecordBook {
        get => _recordBook; set => _recordBook = value;
    }
}
```

View (представление) – уровень отображения. Предоставляет пользователю интерфейс и визуальное отображение данных из модели. Роль представления в классе `Student` будет выполнять файл, который содержит код формы (предположим, что он называется *StudentView.cs*).

Presenter (презентер) – элемент, соединяющий между собой View и Model. View передаёт ему происходящие события, презентер обрабатывает их и при необходимости обращается к Model и возвращает View данные на отрисовку.

Эти элементы можно представить в виде следующей диаграммы:

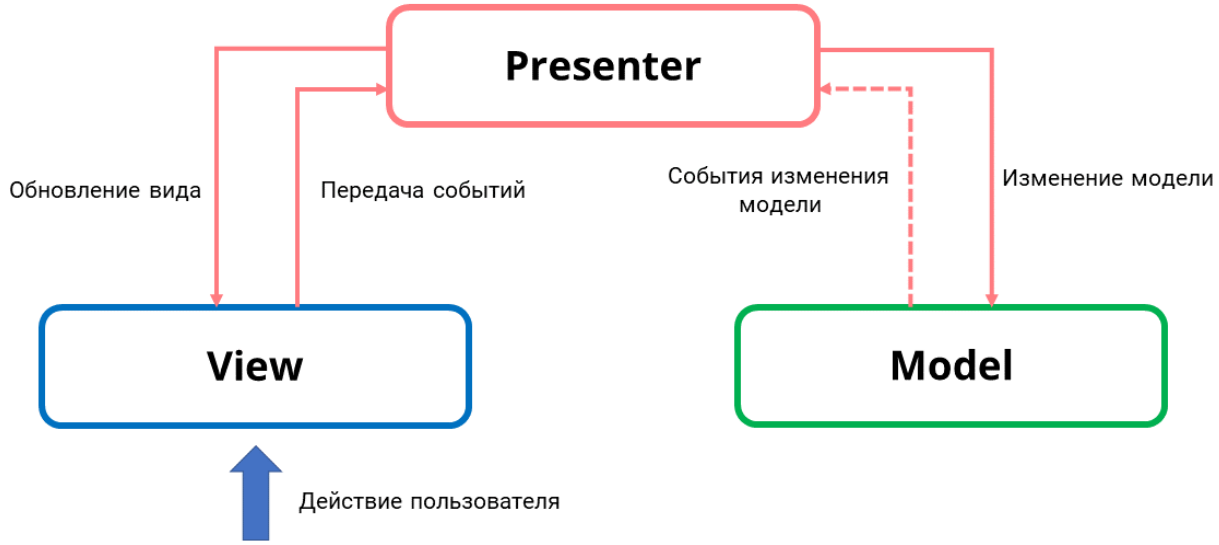

Рисунок 10. Архитектурный паттерн MVP

Каждое представление должно реализовывать соответствующий интерфейс. Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем. Презентер должен иметь ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе.

Листинг 30. Предлагаемый шаблон реализации интерфейса IStudentView

```
public interface IStudentView
{
    string StudentId { get; set; }
    string StudentName { get; set; }
    int StudentRecordBook{ get; set; }
    // And so on...

    bool IsEdit { get; set; }
    bool IsSuccessful { get; set; }

    event EventHandler AddEvent;
    event EventHandler EditEvent;
    event EventHandler DeleteEvent;
    event EventHandler SaveEvent;
    event EventHandler CancelEvent;

    // etc.
}
```

Логика представления *должна иметь ссылку на экземпляр презентера*. Все события представления передаются для обработки в презентер и практически

никогда не обрабатываются логикой представления (в т.ч. создания других представлений).

Листинг 31. Предлагаемый шаблон реализации класса StudentPresenter

```
public class StudentPresenter
{
    private IStudentView _view;
    private IStudentRepository _repository;
    private BindingSource petsBindingSource;
    private IEnumerable<StudentModel> petList;

    public StudentPresenter(IStudentView view, IStudentRepository repository)
    {
        this._view = view;
        this._repository = repository;

        // Connect event handler methods to view events
        this._view.AddEvent += AddStudent;
        this._view.EditEvent += LoadSelectedStudent;
        this._view.DeleteEvent += DeleteStudent;
        //...
    }

    private void AddStudent(object sender, EventArgs e)
    {
        // Some code
    }

    private void LoadSelectedStudent(object sender, EventArgs e)
    {
        // Some code
    }

    private void DeleteStudent(object sender, EventArgs e)
    {
        // Some code
    }

    //...
}
```

Как наверно заметили наиболее внимательные из Вас, тут добавляется ранее не упомянутый интерфейс `IStudentRepository`. Основная его задача – связывать между собой модель более низкого уровня `StudentModel` и модель более высокого уровня `StudentRepository` таким образом, чтобы выполнялся **принцип инверсии зависимостей** (DIP).

Прим. Для того чтобы разобраться в DIP, рекомендуем ознакомиться со статьей на Хабре: https://habr.com/ru/post/313796/

Если сейчас представить нашу программу в виде зависимостей, тогда получится примерно следующая схема:

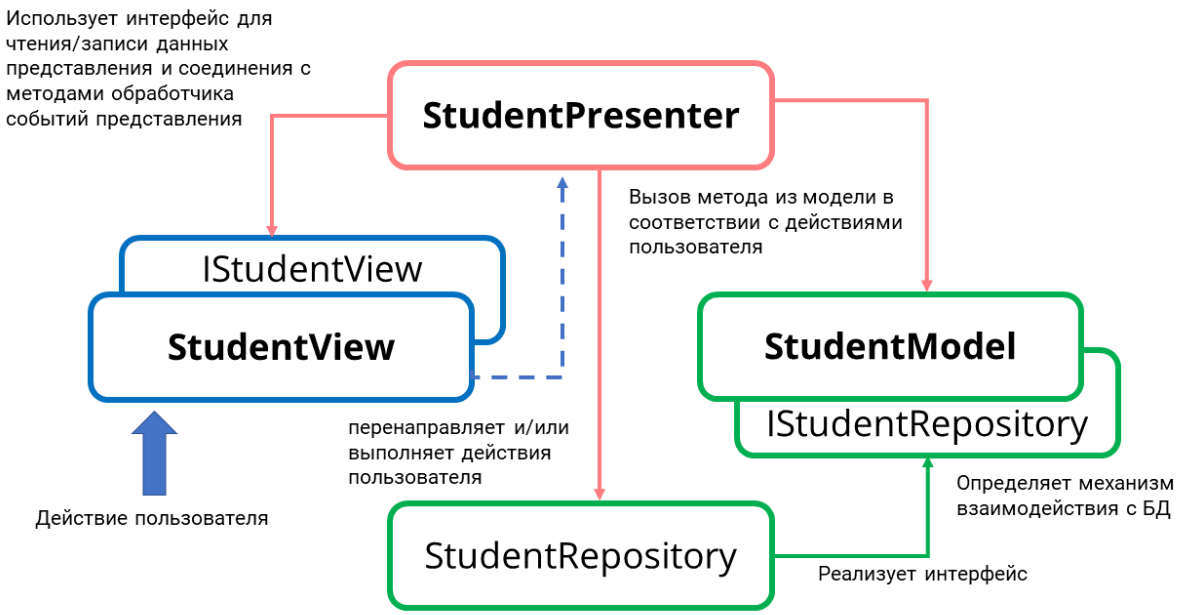

Рисунок 11. Схема взаимодействия компонентов приложения

А вот и примеры класса `StudentRepository` и интерфейса `IStudentRepository`:

Листинг 32. Предлагаемый шаблон реализации IStudentRepository и StudentRepository

```
public interface IStudentRepository
{
    void Add(StudentModel studentModel);
    void Edit(StudentModel studentModel);
    void Delete(StudentModel studentModel);
    IEnumerable<StudentModel> GetAll();
    IEnumerable<StudentModel> Get(int id);
}

public class StudentRepository : IStudentRepository
{
    public StudentRepository() { }

    public void Add(StudentModel studentModel) {
        // Code
    }

    // ...

    public IEnumerable<StudentModel> Get(int id) {
        // Code
    }

    public IEnumerable<StudentModel> GetAll()
    {
        // Code
    }
}
```

Не будем забывать, что наша модель получает все данные из базы данных. В качестве примера взаимодействия с БД реализуем метод `GetAll()`.

Листинг 33. Реализация метода GetAll

```
public IEnumerable<StudentModel> GetAll()
{
    var studentList = new List<StudentModel>();
    // Создаем новое подключение к БД
    using (var connection = new SqlConnection(ConfigurationManager.Connection-
Strings["student"].ConnectionString))
    // Формируем новый SQL запрос
    using (var command = new SqlCommand())
    {
        connection.Open();
        command.Connection = connection;
        command.CommandText = "SELECT * FROM student";
        using (var reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                var studentModel = new StudentModel();
                studentModel.Id = (int)reader[0];
                studentModel.Name = reader[1].ToString();
                studentModel.RecordBook = reader[2].ToString();
                studentList.Add(studentModel);
            }
        }
    }
    return studentList;
}
```

Подключение к БД в C# схоже с Java. В коде выше обращение к строке подключения происходит через класс `ConfigurationManager`. Для того чтобы у вас была возможность обращаться к нему так же, как и в примере выше, необходимо:

- В файле *App.config* (создается вместе с проектом) добавить:

Листинг 34. Добавление строки подключения в App.config

```
…
<configuration>
    …
    <connectionStrings>
        <add name="student"
            providerName="System.Data.ProviderName"
            connectionString="Valid Connection String;" />
    </connectionStrings>
    …
</configuration>
…
```

- Перейти в Project => *“Add Reference…”* и поставить галочку напротив пункта System.Configuration;
- Обратиться к строке подключения в коде при помощи конструкции: `ConfigurationManager.ConnectionStrings["имя_строки"].ConnectionString`.

Задание

- Создать базу данных в MySQL (или в другой СУБД на выбор);

Прим. Рекомендуем ознакомиться с TablePlus – универсальным и мощным инструментом.

- Добавить в БД таблицу Student со свойствами из лаб. №7;
- Реализовать приложение, идентичное приложению из лаб. №7, с использованием MVP и БД.

Задание на доп. баллы

- Реализовать функции динамического поиска и фильтрации;
- Добавить фильтрацию по нескольким полям одновременно;
- Самостоятельно реализовать функцию сортировки номера зачетки по возрастанию / убыванию (встроенный функционал `DataGridView` не считается).

Лабораторная работа 9. Работа с файлами

Цель – формирование навыков чтения и записи файлов различных расширений (*.csv, *.json, *.xml) с использованием C#.

Теоретическая справка

Для чтения и записи файлов в C# традиционно применяются два класса: `StreamReader` и `StreamWriter` соответственно.

Создадим тестовый файл *helloworld.txt*, который будет находиться в директории *C:\Users\имя_пользователя\Documents*

В следующем коде используется класс `StreamReader` для открытия, чтения и закрытия текстового файла. Метод `ReadLine` считывает каждую строку текста и перемещает указатель файла на следующую строку по мере чтения. Если метод `ReadLine` достигает конца файла, он возвращает пустую ссылку.

Листинг 35. Чтение файла построчно с использованием метода ReadLine

```
String? line;

try
{
    StreamReader sr = new
StreamReader("C:\\Users\\mainc\\Documents\\helloworld.txt");
    // Считываем первую строку
    line = sr.ReadLine();
    // Продолжаем чтение пока не достигнем конца файла
    while (line != null)
    {
        // Выводим считанную строку в консоль
        Console.WriteLine(line);
        //читаем следующую строку
        line = sr.ReadLine();
    }

    // Закрываем файл
    sr.Close();
    Console.ReadLine();
}
catch (Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}
```

Кроме того, вся конструкция обернута конструкцией `try…catch`, что позволяет нам поймать ошибки, которые могут теоретически возникнуть при чтении файла. Например, если искомый файл будет отсутствовать в системе.

Естественно, необязательно файл читать файл построчно. Можно считать все его содержимое целиком. Для этого используется метод `ReadToEnd`. Пример представлен ниже:

Листинг 36. Чтение файла целиком с использованием метода ReadToEnd

```
string readContents;
using (StreamReader streamReader = new
StreamReader("C:\\Users\\mainc\\Documents\\helloworld.txt", Encoding.UTF8))
{
    readContents = streamReader.ReadToEnd();
}
```

Второй параметр в конструкторе `StreamReader` задает кодировку файла (параметр по умолчанию – Encoding.UTF8).

Прим. Для чтения также может использоваться `File.ReadAllLines`. При построчном чтении можно использовать оператор итерации `foreach`.

Запись файла тоже не отличается особой сложностью. Класс `StreamWriter` используется для открытия, записи и закрытия текстового файла. Метод `WriteLine` записывает всю текстовую строку в текстовый файл и автоматически добавляет символ возврата каретки или перевода строки (CR/LF). Если перехода на новую строку хочется избежать, тогда можно использовать метод `Write`.

Листинг 37. Запись данных в файл с использованием метода WriteLine

```
try
{
    StreamWriter sw = new StreamWriter("C:\\Test.txt");
    //Запись текста
    sw.WriteLine("Hello World!!");
    //Закрытие файла
    sw.Close();
}
catch (Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}
```

Прим. В качестве материала для самостоятельного изучения почитать про работу с:

1. *XML* – https://goo.su/yzxXOy
2. *JSON* – Изучить работу фреймворка Json.NET и установить его с помощью менеджера пакетов NuGet.

Задание

Разработать программу для чтения/записи файлов в формате *.json, *.xml.

Требуется:

- Каждая запись должна содержать следующие сведения: ФИО студента, номер студенческого билета, направление обучения;

- Записи должны отображаться с использованием элемента управления `ListView`;
- Наличие возможности добавлять и удалять записи при помощи GUI;
- Все пути к файлам должны быть относительными.

Задание на доп. баллы

- Разработать такой же функционал для чтения и записи формата: *.csv.

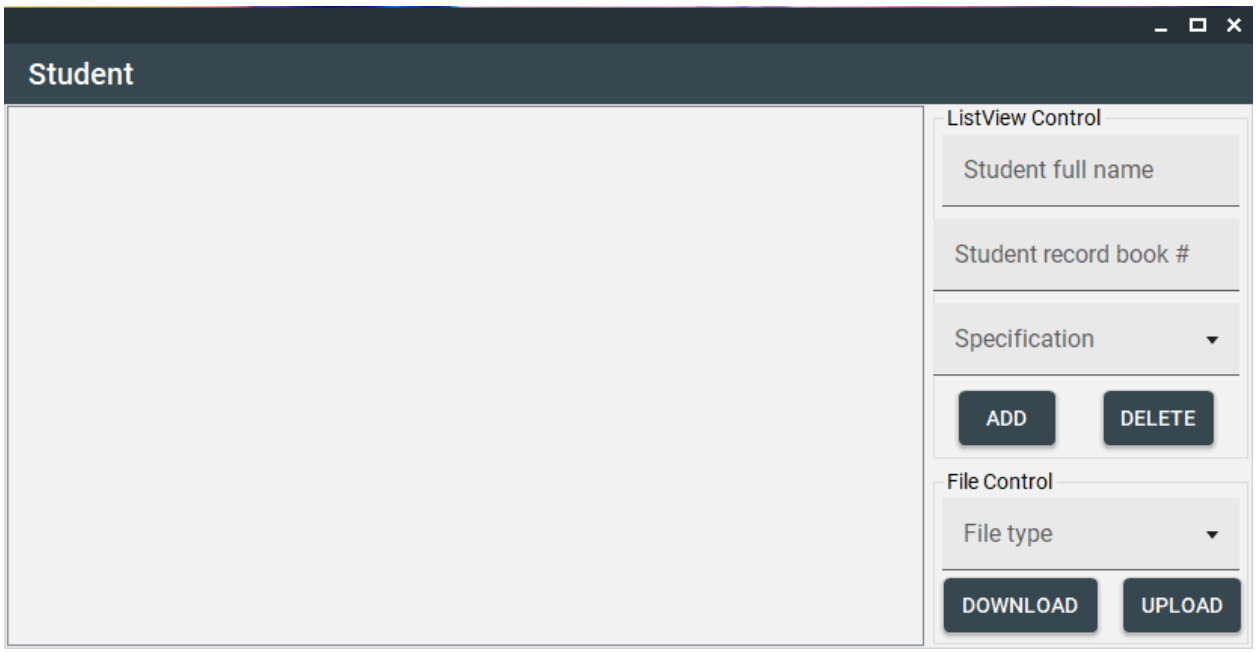

Рисунок 12. Примерный интерфейс программы для работы с файлами *.json и *.xml

Лабораторная работа 10. Создание собственной системы (доп.)

Цель – формирование навыков разработки собственной системы данных, приобретение опыта реализации системы с разграничением прав доступа.

Задание

Разработать демонстрационную БД (рекомендуется взять за основу университет). Продумать систему авторизации. Реализовать RBAC (Role Based Access Control). Добавить личный кабинет пользователя.

Задачи для самостоятельной работы

Задание 1. Комплексные числа

Создайте класс `Complex` для выполнения арифметических действий с комплексными числами. Напишите программу для проверки вашего класса.

Комплексные числа имеют форму RealPart + ImaginaryPart * i, где $i = \sqrt{-1}$.

Используйте переменные с плавающей точкой для представления закрытых полей этого класса. Создайте конструктор, который позволяет задать, действительную и мнимую часть числа. Реализуйте открытые функции-члены для каждого из следующих пунктов:

1) Сложение двух комплексных чисел;
2) Вычитание двух комплексных чисел;
3) Умножение двух комплексных чисел;
4) Печать комплексного числа в форме (a, b), где a — действительная часть, b — мнимая часть числа.

Задание 2. Абстрактные классы

1. Разработать абстрактный класс `Shape` с методами `CalculateArea`, `CalculatePerimeter` и `Print`. Создать производные от `Shape` классы – `Circle`, `Rectangle`, `Triangle`, переопределив методы родительского класса.

2. Разработать абстрактный класс `Animal` с методами печати и проверки, является ли животное живородящим или яйцекладущим. Создать производные от `Animal` классы — `Dog`, `Duckbill`, переопределив методы родительского класса.

Задание 3. Дроби

1. Создать класс `Rational` для выполнения арифметических действий с дробями. Напишите юнит тесты для проверки корректности работы вашей программы.

Используйте целочисленные типы для представления закрытых элементов класса — числителя и знаменателя. Создайте конструктор, который позволяет задать начальные значения числителя и знаменателя. Конструктор должен при необходимости задавать значения по умолчанию и сокращать дроби ($\frac{2}{4}$ как $\frac{1}{2}$). Реализуйте открытые методы для следующих действий (результат должен возвращаться в сокращенной форме):

1) Сложение двух чисел;

2) Вычитание двух чисел;
3) Умножение двух чисел;
4) Деление двух чисел;
5) Проверка на равенство;
6) Печать чисел в формате $a\ /\ b$;
7) Печать чисел в формате с плавающей точкой.

Прим. Предусмотреть, что знаменатель дроби не может равняться 0.

Задание 4. Даты

Создать класс `Date`, с конструкторами и методами: установить дату, увеличить на 1 день (++), уменьшить на один день, добавить дни +=. Перегрузить инкремент и декремент, функцию `ToString`.

Задание 5. Большие числа

Создайте класс `HugeInteger`, который предназначен для хранения целых чисел, содержащих до 40 цифр. Создайте:

1) Методы для ввода, вывода, сложения и вычитания этих чисел;
2) Методы для сравнения этих чисел (для реализации операций =, <>, <, <=, >, >=);
3) Метод для проверки равенства 0 (`IsZero`);
4) Методы для умножения и деления чисел `HugeInteger`.

Задание 5.1. Большие простые числа

Опираясь на класс, описанный в задании 5, реализуйте генератор очень больших простых чисел.

Задание 6. Многочлены

Создать класс `Polinomials` (многочлены) для представления полиномов вида $a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$. Внутренним представлением класса является массив членов. Каждый член состоит из коэффициента и степени члена.

Пример. Член $2x^4$ имеет коэффициент 2 и показатель степени 4

Класс должен содержать: конструктор, деструктор, функции `set` и `get`. Перегрузить:

1) Оператор **+** для сложения полиномов;
2) Оператор **–** для вычитания полиномов;

3) Оператор * для умножения полиномов.

Задание 7. Цифровой счетчик

Цифровой счетчик — это переменная с ограниченным диапазоном, которая сбрасывается, когда ее целочисленное значение достигает определенного максимума.

Пример. Цифровые часы, счетчик километража

Спроектируйте класс подобного счетчика. Обеспечьте возможность установления максимального и минимального значений, увеличения значений счетчика на 1, возвращения текущего значения.

Задание 8. Квадратные матрицы

Создать класс `Matrix` (матрицы) для представления квадратных матриц. Разработайте конструктор и свойства. Перегрузить следующие операторы:

1) Оператор **+** для сложения матриц;
2) Оператор **–** для вычитания матриц;
3) Оператор ***** для произведения матриц.

Задание 8.1. Прямоугольные матрицы

Добавьте в класс `Matrix` поддержку прямоугольных матриц.

Прим. Не забудьте добавить проверку соответствия размеров матриц при выполнении математических операций с ними.

Задание 8.2. Прямоугольные матрицы

Добавьте операции: транспонирования, нахождения определителя и вычисления ранга матрицы.

Задание 9. Логгер

Спроектировать и разработать собственную систему логирования. Требуемый функционал:

1) Реализовать поддержку вывода логов в консоль и записи в текстовый файл *log.txt*;
2) Разделить события на следующие уровни важности: *ERROR*, *WARNING*, *INFO*, *DEBUG* (уровни расположены в порядке уменьшения важности);
3) Выводить сообщения уровень важности которых >= заданному;

Пример. Если установлен уровень важности *INFO*, тогда будут выводиться сообщения с уровнями: *ERROR*, *WARNING* и *INFO*.

4) Добавить возможность отображения времени события.

Пример формата записи лога. [11:44:32] (WARNING) – Вова съел всю кашу.

Задание 10. Генеалогическое древо (доп.)

Разработать программу, предназначенную для формирования и графического отображения (при помощи Windows Forms) генеалогического дерева.

Требования:

1) Наличие полей и функционала для добавления члена семьи;
2) Дружественный и красивый GUI.

Задание 10.1. Продвинутое генеалогическое древо (доп.)

Спроектируйте систему записи и чтения данных. Возможные виды реализации:

1) База данных;
2) Файл формата *.json.

Задание 10.2. Продвинутое генеалогическое древо (доп.)

Убрать возможное ограничение по количеству ветвей.