**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**ГОУ ВПО Кыргызско-Российский Славянский университет**

Хмелёва И.В.

Методические указания к выполнению практических работ по дисциплине

«Объектно-ориентированное программирование»

Бишкек 2023

Оглавление

[Практическая работа № 1 Введение в язык C# 3](#_Toc124421130)

[Задание на практическую работу №1 (8 баллов) 10](#_Toc124421131)

[Практическая работа № 2. Введение в ООП 12](#_Toc124421132)

[Задание на практическую работу №2 (6 баллов) 19](#_Toc124421133)

[Практическая работа №3. Основные принципы ООП 21](#_Toc124421134)

[Задание на практическую работу №3 (8 баллов) 27](#_Toc124421135)

[Вариант 1. Требования к программе 29](#_Toc124421136)

[Вариант 2. Требования к программе 31](#_Toc124421137)

[Вариант 3. Требования к программе 33](#_Toc124421138)

[Практическая работа №4. Абстракция, полиморфизм 34](#_Toc124421139)

[Задание к практической работе №4 (5 баллов). 36](#_Toc124421140)

[Практическая работа №5. Интерфейсы 36](#_Toc124421141)

[Задание на практическую работу №5 (8 баллов) 47](#_Toc124421142)

[Вариант 1. Требования к программе 48](#_Toc124421143)

[Вариант 2. Требования к программе 51](#_Toc124421144)

[Вариант 3. Требования к программе 55](#_Toc124421145)

[Практическая работа №6. Обобщения, коллекции 58](#_Toc124421146)

[Задание к практической работе №6. (8 баллов) 62](#_Toc124421147)

[Вариант 1. 63](#_Toc124421148)

[Вариант 2. 66](#_Toc124421149)

[Вариант 3. 68](#_Toc124421150)

[Практическая работа №7. Делегаты. События. 70](#_Toc124421151)

[Задание на практическую работу №7 (8баллов) 76](#_Toc124421152)

[Вариант 1. Требования к программе 76](#_Toc124421153)

[Вариант 2. Требования к программе 78](#_Toc124421154)

[Вариант 3. Требования к программе 81](#_Toc124421155)

[Дополнительная работа №8 (10 баллов) 83](#_Toc124421156)

[Вариант 1. 84](#_Toc124421157)

[Вариант 2. 86](#_Toc124421158)

[Вариант 3. 88](#_Toc124421159)

[Практическая работа №9. Система ввода-вывода 90](#_Toc124421160)

[Задание на практическую работу №9 (10 баллов) 91](#_Toc124421161)

# Практическая работа № 1 Введение в язык C#

**Краткие сведения о языке C#**

C# является чисто объектно-ориентированным языком. В ООП ход выполнения программы определяется объектами.

Объекты – это экземпляры класса.

Класс – абстрактный тип данных, определяемый пользователем (программистом). Класс может содержать данные (поля) и функции (методы).

В C# запрещены глобальные функции. Все функции должны быть обязательно определены внутри класса. Точка входа программы в C# Main (тут с большой буквы) также должна являться статическим методом класса, как правило, класса Program.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

При создании пустого консольного приложения в Visual Studio будет сгенерирован вот такой шаблон приложения:

using System;

namespace ConsoleApp1

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}}

В начале файла идут директивы using после записываются названия подключаемых пространств имён. Также для каждого приложения по умолчанию создается свое уникальное пространство имен, в котором располагаются все создаваемые программистом элементы.

В VS2022 в среде NetCore все заголовки опущены, т.е. после загрузки среды вы сразу попадаете в точку загрузки основной задачи:

Console.WriteLine("Hello World!");

**Типы данных .NET**

C# является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение. Все типы на платформе .NET делятся на типы значений и ссылочные типы (рис.1).

Типы значений — это типы данных, объекты которых представлены фактическим значением объекта. Если экземпляр типа значения присваивается переменной, то эта переменная получает новую копию значения.

Ссылочные типы — это типы данных, объекты которых представлены ссылкой (аналогичной указателю) на фактическое значение объекта. Если экземпляр ссылочного типа присваивается переменной, то эта переменная будет ссылаться (указывать) на исходное значение. Копирования при этом не происходит.

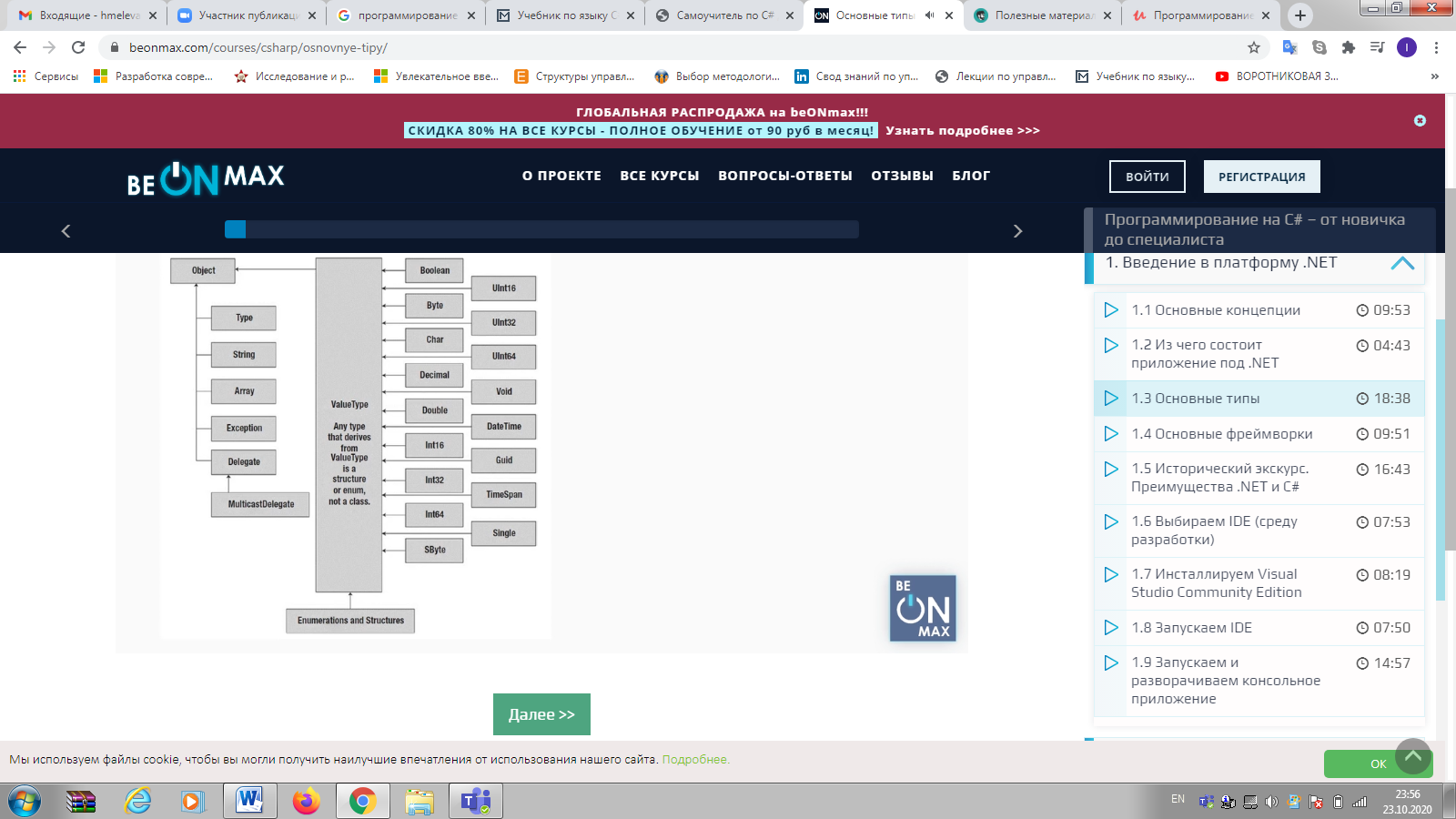


Рис.1. Общая система типов

Базовые типы данных: Boolean, Byte, Double, Int16, Int32, Char, структуры, перечисления.

**Структура** — это тип значения, неявно производный от типа [System.ValueType](https://learn.microsoft.com/ru-ru/dotnet/api/system.valuetype), который, в свою очередь, является производным от типа [System.Object](https://learn.microsoft.com/ru-ru/dotnet/api/system.object). Структуры полезны для представления значений с небольшими требованиями к памяти и для передачи параметров по значению в методы со строгой типизацией параметров.

**Перечисление** — это тип значения, который напрямую наследует от [System.Enum](https://learn.microsoft.com/ru-ru/dotnet/api/system.enum) и предоставляет альтернативные имена для значений базового типа-примитива. Тип перечисления имеет имя, базовый тип, который должен быть одним из встроенных целочисленных знаковых или беззнаковых типов

К ссылочным типам относят: классы, интерфейсы, делегаты, строки

**Объявление переменных**

bool isTrue = true;

int x;

double y = 3.14;

string hello = "Hello Invader!";

char c = 's';

x = 3;

int y = x \* 5;

**Неявная типизация**

C# относится к языкам со строгой типизацией, но допускается операция автоматического выведения типа через использование ключевого слова **var**.

var str = "string";

var c = 32;

Console.WriteLine(str.GetType().ToString());

Console.WriteLine(c.GetType().ToString());

**Ограничения var**

Во-первых, нельзя объявить неявно типизируемую переменную раздельно с инициализацией. Во-вторых, нельзя проинициализировать неявно типизированную переменную значением null.

var c; // ошибка раз

c = 144;

var nl = null; // ошибка два

По арифметическим операциям нет ничего нестандартного. Операторы булевой алгебры также работают по общепринятым сценариям.

**Преобразования типов**

По аналогии с С++ существуют явные и неявные преобразования типов.

double d = 143.234;

int c = d; // неявное приведение типов

int c = (int)d; // явное

**Проверка диапазона**

Есть ключевое слово checked, которое проверяет диапазоны переполнения, и в случае ошибки генерирует исключение.

int a = 53;

int b = 724;

byte c = checked((byte)(a + b));

Console.WriteLine(c);

**Класс Convert**

Для более корректного преобразования типов можно использовать методы класса Convert, которые позволяют приводить практически любые **базовые** **типы** данных друг к другу.

int x = 34;

string sd = "31.45326";

ulong l = 235;

byte b = Convert.ToByte(l);

double d = Convert.ToDouble(sd);

string s = Convert.ToString(x)

**Массивы**

Написание объявлений массивов имеет свои нюансы: квадратные скобки стоят не после имени переменной, а после типа.

int[] mas1;

int[] mas2 = new int[5];

int[] mas3 = new int[5] { 1, 2, 3, 4, 5 };

int[] mas3 = new int[] { 1, 2, 3, 4, 5 };

int[] mas3 = new[] { 1, 2, 3, 4, 5 };

int[] mas3 = { 1, 2, 3, 4, 5 };

**Многомерные массивы**

int[,,] cube = new int[2, 3, 4];

int[,] matrix = new int[2, 3] { { 1, 2, 3 }, { -1, -2, -3 } };

**Массивы массивов**

Нужно отличать многомерные массивы от массива массивов.

Многомерные массивы являются "прямоугольными", а каждый массив внутри массива массивов может иметь свой собственный размер.

int[][] mas = new int[3][];

mas[0] = new int[3];

mas[1] = new int[12];

mas[2] = new int[5];

**Объявление кортежа**

Переменная-кортеж определяется через символы круглых скобок.

(int, int) tuple1;

(int, int) tuple2 = (5, 12);

tuple1 = tuple2;

var tuple3 = (1, "Roger Smith", 45);

**Доступ к элементам кортежа**

Для доступа к элементам кортежа по умолчанию используется имена формата ItemN, где N – номер элемента в кортеже.

var some = (43, "name", true);

Console.WriteLine(some.Item1); // 43

Console.WriteLine(some.Item2); // name

some.Item3 = false;

Console.WriteLine(some.Item3); // False

Кортежи являются строго типизированными, и элемент одного типа нельзя заменить элементом другого типа. Вместо ItemN можно назначать элементам кортежа свои наименования, при помощи нотации двоеточия :

Причём это работает только при использовании автоматического выведения типов с var.

// (string, int, double) person = ("Tom", 25, 81.23); // Так бы не работало

var person = (name: "Tom", age: 25, weight: 81.23);

person.age++;

person.weight = 88.3;

person.name = Console.ReadLine();

**Раскрытие кортежей**

Кортеж можно присвоить группе переменных, чтобы разделить его значения.

var (name, age) = ("Tom", 23);

Console.WriteLine(name); // Tom

Console.WriteLine(age); // 23

Console.Read();

Основное применение кортежей - возврат сразу нескольких значений из метода.

**Цикл foreach**

Помимо стандартного три циклов for, while, do-while в C# присутствует цикл foreach, разработанный в целях упрощения перебора элементов каких-либо последовательностей.

Цикл имеет следующую форму записи:

foreach (тип\_данных имя in контейнер)

{

действия

}

Пример перебора элементов в одномерном и двухмерном массивах:

int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

foreach (int element in arr)

{

Console.WriteLine(element);

}

int[][] mtr = new int[3][];

mtr[0] = new[] { 1, 2, 3 };

mtr[1] = new[] { 4, 5, 6, 7 };

mtr[2] = new[] { 8, 9 };

foreach (int[] array in mtr)

{

foreach (int element in array)

{

Console.WriteLine(element);

}

}

**Методы массива**

Массив представляет собой объект агрегирующего класса. Как любой класс, он содержит методы. Их существует огромное число, вот лишь некоторые из них:

|  |  |
| --- | --- |
| Average() | Высчитывает среднее значение для элементов массива |
| CopyTo() | Копирует все элементы этого массива в другой |
| GetType() | Позволяет получить тип текущего массива |
| GetLength() | Получить количество элементов в массиве |
| Rank() | Получает количество измерений этого массива |
| Min() | Выдает минимальное значение элемента массива |
| Max() | Выдает максимальное значение элемента массива |

Пример- сортировка массива

static void Main(string[] args)

{

int[] nums = new int[7];

Console.WriteLine("Введите семь чисел");

for (int i = 0; i < nums.Length; i++)

{

Console.Write("{0}-е число: ", i + 1);

nums[i] = Int32.Parse(Console.ReadLine());

}

int temp;

for (int i = 0; i < nums.Length - 1; i++)

{

for (int j = i + 1; j < nums.Length; j++)

{

if (nums[i] > nums[j])

{

temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

}

Console.WriteLine("Вывод отсортированного массива");

for (int i = 0; i < nums.Length; i++)

{

Console.WriteLine(nums[i]);

}

Console.ReadLine();

Если использовать метод сортировки объекта ARRAY, то процедура упрощается: Array.Sort(nums);

Теперь в массиве nums содержится отсортированный массив.

Следующий пример показывает использование методов объекта ARRAY для создания и инициализации массива:

Array myIntArray=Array.CreateInstance( typeof(Int), 5 );

for (int i = myIntArray.GetLowerBound(0); i <= myIntArray.GetUpperBound(0); i++ )

myIntArray.SetValue( i+1, i );

**Строки**

В C# строки являются объектами класса String. Содержимое объекта String не подлежит изменению, однако это не сказывается на удобстве его использования.

Конкатенация (объединение) строк представляет из себя тривиальную операцию +

string str1 = " is ";

string res = "This" + str1 + "string" + "!";

Для создания строк из переменных можно воспользоваться специальным префиксным знаком $:

int[] mas = { 1, 2, 3 };

string str = mas[1].ToString();

string st = $"Element {mas[1]} has type: {str} !\n";

Console.Write(st);

**Операции со строками**

// Сравним первые две строки

string s1 = "это строка";

string s2 = "это текст, а это строка";

if (String.CompareOrdinal(s1, s2) != 0)

Console.WriteLine("Строки s1 и s2 не равны");

if (String.Compare(s1, 0, s2, 13, 10, true) == 0)

Console.WriteLine("При этом в них есть одинаковый текст");

// Конкатенация строк

Console.WriteLine(String.Concat("\n" + "Один, два ", "три, четыре"));

// Поиск в строке

// Первое вхождение подстроки

if (s2.IndexOf("это") != -1)

Console.WriteLine("Слово \"это\" найдено в строке, оно " +

"находится на: {0} позиции", s2.IndexOf("это"));

**Параметры методов**

Существует два основных способа передачи параметров в метод.

* по ссылке;
* по значению.

Объекты классов по умолчанию передаются по ссылке, а базовые типы данных и структуры - по значению.

**Модификатор ref**

При передаче параметров по ссылке перед параметрами используется модификатор ref:

static void Main(string[] args)

{

int x = 10;

int y = 15;

Addition(ref x, y); // вызов метода

Console.WriteLine(x);

Console.ReadLine();

}

// определение метода

static void Addition(ref int x, int y)

{

x += y;

}

**Модификатор out**

Модификатор out позволяет через параметры метода возвращать значение. Такое часто применяется, требуется вернуть более одного значения различных типов.

static void Main(string[] args)

{

int x = 10;

int z;

Sum(x, 15, out z);

Console.WriteLine(z);

Console.ReadLine();

}

static void Sum(int x, int y, out int a)

{

a = x + y;

}

Массивы могут передаваться в методы в качестве параметров, а также возвращаться из методов. Для возврата массива достаточно объявить массив как тип возврата.

// Данный метод возвращает массив Info

static Array myReturn()

{

string[] Info = { "Name", "Family", "Adress" };

return Info;

}

// Передача массива в метод в качестве параметра

static int mySort(Array sortArray, int i)

{

Array.Sort(sortArray);

return (int)sortArray.GetValue(i);

}

static void Main(string[] args)

{

foreach (string s in myReturn())

Console.WriteLine(s);

int[] mySortArray = { 4, 1, 2, 5, 3 };

for (int i = 0; i <= 4; i++)

mySortArray[i] = mySort(mySortArray, i);

Console.WriteLine("\nОтсортированный массив:\n");

foreach (int i in mySortArray)

Console.Write("\t" + i);

Console.ReadLine();

}

**Кортежи и методы**

Как и любой тип, кортежи могут использоваться в методах в качестве параметров и возвращаемых значений.

Например:

static void Main(string[] args)

{

var tuple = GetValues();

Console.WriteLine(tuple.Item1); // 1

Console.WriteLine(tuple.Item2); // 3

Console.Read();

}

private static (int, int) GetValues()

{

var result = (1, 3);

return result;

}

**или**

static void Main(string[] args)

{

var tuple = GetNamedValues(new int[] { 1, 2, 3, 4, 5, 6, 7 });

Console.WriteLine(tuple.count);

Console.WriteLine(tuple.sum);

Console.Read();

}

private static (int sum, int count) GetNamedValues(int[] numbers)

{

var result = (sum: 0, count: 0);

for (int i = 0; i < numbers.Length; i++)

{

result.sum += numbers[i];

result.count++;

}

return result;

}

## Задание на практическую работу №1 (8 баллов)

Работа состоит из 4 заданий, которые направлены на освоение основ языка программирования C#.

**Задание 1** (2 балла) выполняется по вариантам, согласно № в групповом журнале.

1. Дана строка, состоящая из групп нулей и единиц. Посчитать количество нулей и единиц.

2. Посчитать в строке количество слов.

3. Найти количество знаков препинания в исходной строке.

4. Дана строка символов. Вывести на экран цифры, содержащиеся в строке.

5. Дана строка символов, состоящая из произвольных десятичных цифр, разделенных пробелами. Вывести количество четных чисел в этой строке.

7. Дана строка символов. Вывести на экран количество строчных русских букв, входящих в эту строку.

8. Дана строка символов. Вывести на экран только строчные русские буквы, входящие в эту строку.

9. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. В каждом слове заменить первую букву на прописную.

10. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. Удалить первую букву в каждом слове.

11. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. Поменять местами i- и j-ю буквы. Для ввода i и j на форме добавить свои поля ввода.

12. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. Поменять местами первую и последнюю буквы каждого слова.

13. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. Заменить все буквы латинского алфавита на знак ‘+’.

14. Дана строка символов, содержащая некоторый текст на русском языке. Заменить все большие буквы буквы ‘А’ на символ ‘\*’.

15. Дана строка символов, содержащая некоторый текст. Разработать программу, которая определяет, является ли данный текст палиндромом, т.е. читается ли он слева направо так же, как и справа налево ( например, «А роза упала на лапу Азора»).

16. Дана строка символов, состоящая из произвольного текста на английском языке, слова разделены пробелами. Сформировать новую строку, состоящую из чисел длин слов в исходной строке.

**Задания 2-4 общие для всех вариантов (по 2 балла)**

2. Создайте массив из 10 элементов. Затем сформируйте второй массив, каждый i-ый элемент которого должен быть равен сумме i-ого элемента отсортированного по убыванию массива и оригинального i-ого элемента.

3. Разработайте метод, возвращающий целочисленный массив элементов со значениями, переданными в качестве набора параметров. Метод первым параметром должен принимать массив, а дальше переменное количество чисел, которые будут удаляться из этого массива. Возвращаться должен новый массив, состоящий из элементов оригинального массива, за исключением удаляемых.

4. Напишите метод, принимающий параметром массив чисел, и который будет возвращать минимальное значение, максимальное значение и среднее значение среди положительных элементов массива. Необходимо сделать это двумя способами: через out параметры и при помощи кортежа.

# Практическая работа № 2. Введение в ООП

Цель: овладение навыками проектирования простейших классов и создания объектов класса; освоение принципов инкапсуляции и наследования.

**Введение в ООП**

Объектно-ориентированное программирование – подход к написанию программ, когда мы оперируем не базовыми элементами – числами, строками, символами, а построенными на их основе абстрактными высокоуровневыми сущностями. Главное в объектно-ориентированном программировании – архитектура, т.е. проектирование кода. Вы не просто пишите программу как последовательность действий и набор функций, а проектируете элементы системы, объединяете логически связанные компоненты, описываете принципы их взаимодействия. ООП – это не просто написание программ. Это их моделирование и структурирование.

Что важно понять для работы с ООП, – это то, что все типы являются классами.

Ваша программа описана в классе программы, для работы с консолью используется класс консоли, для работы с окнами – класс окна, для работы с лохматым померанским шпицем – класс собачки.

Класс – это абстрактный тип данных, определяемый программистом.

Объекты – это экземпляры класса, т.е. переменные типа класса.

// Объявление класса

class ИмяКласса

{

// члены класса

}

Члены класса – это данные и функции для работы с этими данными.

Имя класса можно считать именем нового типа данных.

Класс можно считать "чертежом", "схемой" какого-либо объекта. Класс сам по себе является просто описанием какой-либо сущности.

Объект (экземпляр) же является фактическим представлением этой сущности.

Например, Boeing777 – класс, представляющий одноимённый самолёт. Но это еще не какой-либо конкретный самолёт этой модели. Можно создать экземпляры этого класса boeing1, boeing2 и boeign3, которые уже будут представляться реальные воплощения, 3 разных самолётов такой модели.

Создание объекта класса осуществляется с помощью оператора new:

ИмяКласса имяОбъекта = new ИмяКласса();

Random rand = new Random();

DateTime dt = new DateTime(1991, 10, 23);

string[] mas = new string[4];

List<int> list = new List<int>();

Класс содержит:

* описание объекта- элементы данных (поля класса);

описание поведения объекта- функциональные элементы (функциональные элементы подразделяются на: методы; свойства; конструкторы; операторы; финализаторы; индексаторы).

* Перечень полей *обычно* статический, то есть зафиксированный в определении класса; перечень текущих значений полей *обычно* динамический, то есть может меняться после создания объекта, так как во всех языках есть механизмы, позволяющие ограничить изменение полей после создания объекта.

Перечень методов, *обычно* статический; результаты вызова методов зависят от текущего состояния или, по-другому, определяются историей объекта (то есть историей изменения его состояния), так как один и тот же метод в общем случае может возвращать разные результаты в зависимости от состояния.

Элементы данных подразделяются на:

* поля;
* константы;
* события.

Поле – основной член класса – переменная внутри класса, содержащая некоторое значение.

Константы – это неизменяемые поля. Объявляются с помощью ключевого слова **const**. Константа не может быть объявлена без указания значения.

// Описание класса Dog

class Dog

{

// Поля этого класса

public string name;

public string breed;

public int age;

public string owner;

public const int months = 12;

// Открытый метод

public void Walk()

{

Console.WriteLine("The {0} walks!", this.name);

Random randGen = new Random();

int rndNumber = randGen.Next(4);

if (rndNumber == 3)

this.Pee();

}

// Скрытый метод

private int Pee()

{

Console.WriteLine("The {0} has urinated..", this.name);

Random randGen = new Random();

int strenth = randGen.Next(1, 11);

return strenth;

} }

static void Main(string[] args)

{

// Создание экземпляра класса Dog с именем dg1

Dog dg1 = new Dog();

// Присваивание значений полям

dg1.name = "Альфред";

dg1.breed = "Бернский зенненхунд";

// Создание массива под 4 объекта класса Dog

Dog[] flock = new Dog[4];

// Во второй элемент массива записываем ранее созданный объект

flock[2] = dg1;

// Вызываем для него метод Walk

flock[2].Walk(); }

**Ключевое слово this**

Для доступа к членам текущего экземпляра класса можно использовать ключевое слово this. Это ключевое слово предоставляет ссылку на текущий объект класса. В принципе, использовать this не обязательно, это бывает нужно при совпадении имен. Например:

// Допишем новый метод ChangeOwner

// который принимает параметр: строку owner

public bool ChangeOwner(string owner)

{

// Сравнивается поле owner в текущем (this) объекте класса, со значением параметра owner

if (this.owner == owner)

{

return false;

}

// В поле записывается значение из параметра

this.owner = owner;

return true;

}

**Спецификаторы доступа**

Уровнем доступности каждого элемента класса можно управлять. Это делается с помощью спецификаторов доступа:

* public – общедоступный член класса;
* private – член класса доступен только внутри данного класса;
* protected – член класса доступен только внутри данного класса и его производных классов;
* internal – член класса доступен только внутри данной сборки (~программы).

**Методы**

Основой любого класса являются методы – функции внутри класса, выполняющие какие-либо действия над полями этого класса.

**Статические члены класса**

Обычные методы и вообще всё, описанное внутри класса, существует в контексте какого-либо объекта этого класса. Т.е. для каждого экземпляра класса действуют свои копии методов и полей.

Можно создать элементы класса, которые будут относиться непосредственно к самому классу, а не к объектам. Такие элементы класса называется статическими, являются общими для всех объектов и существуют в единственном экземпляре.

Чтобы объявить такой элемент, нужно к его описанию добавить ключевое слово static.

**Статические методы**

Например, как происходит работа с консолью в C#?

Console.WriteLine("message");

string input1 = Console.ReadLine();

Console.Write("end;\n");

Обратите внимание, здесь нигде не создается объект класса Console, мы напрямую обращаемся с именем класса. Посмотрим на различия между вызовом статических и обыкновенных методов:

class MinMax

{

// Статический метод

public static int Min(int x, int y)

{

int z = (x < y) ? x : y; // Тернарный оператор

return z;

}

// Обычный метод

public int Max(int x, int y)

{

int z = (x > y) ? x : y;

return z;

}

}

class Program

{

static void Main(string[] args)

{

int a = -2;

int b = 23;

Console.WriteLine("a={0}, b={1}", a, b);

// Вызов статического метода для класса MinMax

int k = MinMax.Min(a, b);

Console.WriteLine("Min = {0}", k);

MinMax instance = new MinMax();

// Вызов обычного метода для объекта instance класса MinMax

instance.Max(a, b);

}

}

Статические методы применяются повсеместно, и используются в тех случаях, когда пользователю нет необходимости создавать объект для работы с этим методом.

Статическими, например, могут быть какие-нибудь константы внутри класса, общие для всех объектов. Или методы, никак не затрагивающие всей основной структуры класса и работающие обособленно.

**Свойства**

Для создания большей гибкости и безопасности, внутренних полей, были разработаны такие элементы класса как Свойства.

Свойством в C# называются специальные методы доступа (чтения/записи) к полю класса (get и set). Описание свойства имеет следующий синтаксис:

static void Main(string[] args)

{ // Закрытое поле

private int name;

// Открытое свойство для этого поля

public int Name

{ // Геттер - чтобы получить значение

get

{

return this.name;

}

// Сеттер - чтобы присвоить значение

set

{

this.name = value; // value - специальное ключевое слово, описывающее присваиваемое значение

}

Свойство состоит из двух частей:

* метод-аксессор (геттер) get;
* метод-аксессор (сеттер) set.

Метод get обязательно должен возвращать значение.

В методе set существует неявный параметр value, в котором хранится присваиваемое значение.

Поле и связанное с ним свойство – разные элементы, поэтому должны иметь разные имена. Общепринято называть Свойство так же как называется Поле, только с заглавной буквы.

Поля в классе принято всегда делать приватными, т.е. скрытыми.

И для того, чтобы предоставить доступ к полю, используется Свойство – публичный элемент, через который пользователь читает (get) или записывает (set) значения в приватное поле.

Т.е. свойство является посредником между пользователем и данными, и когда нужно обращаться к полю всегда нужно делать это через свойство.

В свойствах можно описывать необходимую логику или форматирование для значений.

Один из аксессоров можно опускать. Если написать Свойство только с методом get, то свойство будет только для чтения: из него можно будет получить значение, а записать нельзя.

И менее распространённый вариант, с одним методом set, свойство будет только для записи: в него можно записывать значение, но нельзя получать.

При необходимости, внутри свойства можно задавать какие-нибудь проверки, или в принципе писать любой нужный код.

// Закрытое поле

private string name;

// Открытое свойство для него

public string Name

{

// Геттер с логикой

get

{

// Выдаёт не просто значение из поля, но с припиской "Name is "

string res = "Name is " + this.name;

return res;

}

// Сеттер с логикой

set

{

// Записывает значение только если оно не равно пустой строке или нулевой ссылке

if (value != "" && value != null)

name = value;

}

**Конструктор** – особый метод, вызываемый при создании экземпляра класса. Он называется так же, как класс, и не имеет типа возвращаемого значения.

В классе всегда должен быть хотя бы один конструктор (если его нет – он создаётся неявно сам).

Выделяют 4 основных типа конструкторов:

* конструктор по умолчанию;
* конструктор с аргументами;
* конструктор копирования;
* статический конструктор.

Конструктор, объявленный без аргументов, называется конструктором по умолчанию и должен присутствовать в любом классе.

Если в классе не указать ни одного конструктора, компилятор сам создаст конструктор по умолчанию на этапе компиляции.

// Класс без конструктора (явного)

class A

{

// С открытыми полями

public int x, y, z;

}

// Класс с конструктором

class B

{

// С открытыми полями

public int x, y, z;

// Конструктор по умолчанию - без параметров

public B()

{

// Инициализация полей

x = 3;

y = 4;

z = 5;

}

}

class Program

{

public static void Main(string[] args)

{

// При создании объекта через new вызывается конструктор

A a1 = new A(); //x = y = z = 0;

B b1 = new B(); //х= 3, y = 4, z = 5 } }

**Конструкторы с аргументами**

Конструкторы могут принимать различные аргументы, с помощью которых объекты классов инициализируются разнообразными способами

class D

{

// Открытые поля

public int x, y, z;

// Конструктор по умолчанию

public D()

{

// Инициализация всех полей

x = y = z = 0;

}

// Конструктор с двумя параметрами

public D(int x, int y)

{

// Инициализация всех полей

this.x = x;

this.y = y;

this.z = x + y;

}

// Конструктор с тремя параметрами

public D(int x, int y, int z)

{

// Инициализация всех полей

this.x = x;

this.y = y;

this.z = z;

}

}

class Program

{

public static void Main(string[] args)

{

// Создаём объект класса D используя конструктор по умолчанию, x = y = z = 0

D did = new D();

// Создаём объект класса D используя конструктор с 2 параметрами, x = 5, y = 8, z = 13

D opo = new D(5, 8);

// Создаём объект класса D используя конструктор с 3 параметрами, x = 7, y = 2, z = 6

D fer = new D(7, 2, 6);

}

class ClassC

{

// Три открытых поля

public int x, y, z;

// Конструктор по умолчанию

public ClassC()

{

// Инициализирует все поля значением -1

x = y = z = -1;

}

// Конструктор копирования

public ClassC(ClassC obj\_c)

{

// Копирует все данные из объекта, переданного параметром obj\_c, в текущий объект this

this.x = obj\_c.x;

this.y = obj\_c.y;

this.z = obj\_c.z;

} }

Необходимость в конструкторе копирования обуславливается тем, что класс является объектом ссылочного типа, а это означает, что после операции присваивания между двумя объектами класса произойдет копирование ссылки на один и тот же объект, а не создание объекта-копии.

// Создаём объект класса ClassC obj1

ClassC obj1 = new ClassC();

// Здесь не создаётся новый объект obj2

ClassC obj2 = obj1;

// obj1 и obj2 указывают на один и тот же объект класса

obj2.x = -10; // obj1.x == -10

// А вот здесь создаётся новый объект - копия объекта obj2

ClassC obj3 = new C(obj2);

// obj3 имеет такие же значения полей как и obj2, но является другим объектом

obj3.x = 2000; // obj2.x == -10

Статический конструктор используется для инициализации статических элементов класса, и не может иметь параметров. Он неявно вызывается средой CLR сам, до момента первого использования класса.

class SimpleClass

{ // Статическое поле только для чтения

static readonly long baseline;

// Статический конструктор

static SimpleClass()

{// Который инициализирует статическое поле

baseline = DateTime.Now.Ticks;

}

Для любого класса желательно всегда определять конструктор умолчанию и конструктор копирования, так как они являются универсальными и могут требоваться некоторыми встроенными алгоритмами.

**Финализатор** – специальный метод, который вызывается перед тем, как Сборщик Мусора соберется удалить ваш объект. Он должен называться так же, как и сам класс, только перед именем должен стоять знак тильды ~.

Поскольку в C# работает сборщик мусора, то финализатор используется довольно редко, он оставлен в языке для случаев, когда при работе программы необходимо удалить объект, занимающий много места в памяти, и задолго до окончания работы приложения, когда начнет работать сборщик мусора.

## Задание на практическую работу №2 (6 баллов)

Напишите статический метод, принимающий параметр-строку. Метод должен возвращать число, полученное путём следующих вычислений: сумма кодов всех уникальных символов в строке в степени количества их повторений, и всё это по модулю 8.

Например, для строки «Мурсапилами» нужно посчитать:

Коды: М – 1052, у – 1091, р – 1088, с – 1089, а – 1072, п – 1087, и – 1080, л – 1083, м – 1084.

(1052^1 + 1091^1 + 1088^1 + 1089^1 + 1072^2 + 1087^1 + 1080^2 + 1083^1 + 1084^1) % 8 = (1052 + 1091 + 1088 + 1089 + 1149184 + 1087 + 1166400 + 1083 + 1084) % 8 = 2323158 % 8 = 6.

2. Разработайте класс, который должен содержать:

* элементы разного уровня доступа (public и private);
* не менее 4 свойств;
* не менее 3 методов;
* хотя бы одну перегрузку метода;
* статический метод;
* константное или поле только для чтения;
* не менее 3 конструкторов;
* финализатор.

В Main-е создайте объекты вашего класса и продемонстрируйте работоспособность конструкторов и методов. Проверьте, как работает операция присваивания одного объекта другому.

Тематику класса выберите из таблицы в соответствие с вариантом. Ваш вариант – это число, которое вернёт метод из первого задания для строки с вашей фамилией.

|  |  |
| --- | --- |
| Вар. | Задание |
| 0 | Класс – кнопка (как элемент управления в какой-нибудь программе, а не та кнопка, которую на стулья кладут). Примеры свойств: расположение, цвет, текст. Примеры методов: Нажать, Рассчитать новое положение, Заблокировать. |
| 1 | Класс – вертолет (допустим, в симуляторе для обучения пилотов). Примеры свойств: модель, грузоподъемность, мощность двигателя. Примеры методов: Заправиться, Полететь в точку A, Отправиться на посадку. |
| 2 | Класс – четырёхугольная пирамида (для 3d моделирования). Примеры свойств: массив опорных точек, цвет, прозрачность. Примеры методов: Рассчитать объем, Повернуть на, Сбросить цвет. |
| 3 | Класс – герой (для какой-нибудь игры). Примеры свойств: имя, уровень, специализация. Примеры методов: Взять предмет, Поделиться экипировкой, Перейти в локацию. |
| 4 | Класс – автомобиль (например, для салона продажи). Примеры свойств: мощность, количество пассажирских мест, модель. Примеры методов: Перекрасить, Списать, Заказать у производителя. |
| 5 | Класс – работник (для системы управления предприятием). Примеры свойств: идентификатор, стаж, отдел. Примеры методов: Перевод в отдел, Повысить, Получить сводку. |
| 6 | Класс – пользователь (например, на каком-нибудь сайте). Примеры свойств: имя пользователя, электронная почта, подписка на рассылку. Примеры методов: Заблокировать, Получить информацию об активности, Привязать телефон. |
| 7 | Класс – банковский перевод (внутри банковской системы). Примеры свойств: получатель, отправитель, комиссия. Примеры методов: Зафиксировать, Отклонить, Сформировать выписку. |

# Практическая работа №3. Основные принципы ООП

Цель: познакомиться с основными принципами ООП- абстракция, наследование и инкапсуляция. Научиться применять свойства, индексаторы, создавать массивы различного типа: одномерные, двумерные и ступенчатые

**Принципы ООП**

* абстракция;
* инкапсуляция;
* наследование;
* полиморфизм

**Абстракция данных** – это принцип описания программных компонентов с привязкой к конкретной предметной области и чётким определением их концептуальных границ.

Элементы программы разделяется на уровни абстракции, что позволяет работать с объектами, не вдаваясь особенности их реализации, и упрощает их использование в прикладном коде.

Допустим, у вас сервис по продаже автомобилей. И чтобы было удобнее работать на уровне целого автомобиля, вы описываете абстракцию машины, внутри которой уже будет содержаться информация о всех его комплектующих: двигателе, коробке передач, электронике, тормозной системе, амортизации и т.д.

**Инкапсуляция** – это концепция, согласно которой любой класс должен рассматриваться как «чёрный ящик», т.е. программный объект, использующий класс, должен видеть только интерфейс класса, имея к нему доступ, и не вникать в его внутреннюю реализацию. Поэтому данные в классе программируются таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а через методы класса. Инкапсуляция объединяет данные и методы в одном типе, ограничивая доступ к хранимой информации и скрывая программную реализацию. Она позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов. Программисты, разрабатывающие класс, могут реализовывать его свойства и методы, не заботясь о том, как другие программисты будут их использовать, а программисты, использующие класс, не должны заботиться о реализации свойств и методов класса.

Взаимодействие программистов осуществляется по интерфейсу, т.е. набору методов с чётко определенной сигнатурой вызова и возвращаемым значением. Инкапсуляция отражает такую особенность предметной области, как несущественность внутренней структуры объектов при их взаимодействии с другими объектами. Эта концепция предписывает программистам использовать готовые классы в своих программах только на основе их описаний, не вдаваясь в детали их реализации.

**Индексаторы**

Индексаторы позволяют обращаться с объектом класса так, как будто он является массивом, т.е. применять оператор квадратных скобок.

Индексатор по синтаксису напоминает свойство. Он также состоит из методов-аксессоров get и set, только в дополнение принимает параметр – индекс.

// Пример индексатора

public тип\_возвращаемого\_значения this[тип\_индекса j] // Принимает "параметр" - индекс

{

get

{

// Действия с return

}

set

{

// Установка значения c value

}

Обычно индексаторы используются в классах, которые содержат массив или другую коллекцию, для простого доступа к их элементам

class MyArray

{ // Приватное поле - массив байт

private byte[] innerArr = new byte[100];

// Индексатор к текущему классу, для доступа к элементам закрытого массива

public byte this[int i]

{ // Геттер, возвращающий элемент массива по переданному индексу i

get

{

return innerArr[i];

}

// Сеттер, записывающий значение в элемент массива по индексу i

set

{

innerArr[i] = value;

}

} }

class Program

{

static void Main()

{

// Создаём экземпляр класса MyArray

var arr = new MyArray();

// Записываем значение 255 по индексу 0 во внутреннее поле-массив класса через индексатор

arr[0] = 255;

// Обращаемся к нулевому элементу массив через индексатор

Console.WriteLine(arr[0]);

}

Как и методы, индексаторы могут быть перегружены. Т.е. в одном классе может быть сразу несколько индексаторов, различающихся типами индексов и их количеством. Индексатор может принимать не только один индекс, но и сразу несколько, как при использовании многомерных массивов.

**Наследованием** называется возможность абстрактной сущности перенимать данные и функциональность некоторой другой сущности, способствуя повторному использованию программного кода.

Так, начиная с самых абстрактных понятий и сущностей, может формироваться иерархия классов, переходя от полной абстракции к частным случаям реализации.

Другими словами, можно создавать классы по шаблону других классов, т.е. чтобы они сразу были не пустыми, а уже содержали какую-то начальную логику, от которой начиналась выстраиваться их собственная.

Класс, от которого наследуется класс, называется базовыми или родительским. Новый класс, создаваемый на основе базового класса, называется производным классом. Производный класс перенимает всю функциональность (свойства/методы) базового класса, которая по необходимости может быть переопределена. В производный класс могут быть добавлены любые новые члены, как в любой другой класс. Таким образом, производные классы позволяют расширять функциональность базовых классов, без их изменения. Главное назначение механизма наследования – повторное использование кода. А также полиморфизм подтипов. Вообще, в ООП наследование бывает одиночным и множественным.

* одиночное наследование – когда у производного класса может быть только один базовый класс;
* множественное наследование – когда один класс может быть производным сразу от нескольких других классов.

В C# разрешено только одиночное наследование.



Рисунок 2. Транзитивность

**Наследование транзитивно**

Это значит, что любые производные классы от производного какого-либо класса, также являются производными от него. И в обратную сторону, все классы по иерархии наследования выше текущего класса являются для него базовыми.

Т.е. если класс A потомок класса B, а B потомок класса C, то A тоже будет считаться потомком C. Наследование обычно предполагает уточнение какой-либо сущности.

Самый первый класс в иерархии наследования является самым абстрактным и обобщённым, а последующие всё больше и больше уточняют поведение и подходят к частным случаям реализации.



Рисунок 3. Проектирование классов

Для того чтобы объявить, что класс является производным от какого-то другого класса, нужно при его описании после имени класса через двоеточие (:) написать имя родительского класса.

// Какой-то класс

class SomeClass

{

// ...

}

// Класс DerivedClass - производный от класса SomeClass

class DerivedClass : SomeClass

{

// ...

}

Производный класс наследует всю функциональность базового класса, но не всей ей может пользоваться.

Private члены базового класса не доступны в производных.

Если нужно описать закрытый элемент класса, который должен быть виден и потомкам, используется спецификатор доступа protected



Рисунок 4. Особенности спецификаторов доступа

class Gadget

{

// Открытое свойство

public string Name { get; set; }

// Закрытое поле

private int gadgetId;

// Защищённый метод

protected void UpdateNameVersion()

{

// Дописываем к имени окончание

Name += "ver. 2.0";

}

}

// Производный класс от гаджета

class Smartphone : Gadget

{

// Конструктор по умолчанию

public Smartphone()

{

// Обращаемся к разным элементам базового класса

Name = "Vivo Nex 7";

gadgetId = 2; // Не доступно

UpdateNameVersion();

}

}

class Program

{

static void Main(string[] args)

{

// Создаём объект класса смартфон

Smartphone sm = new Smartphone();

// Обращаемся к разным членам класса

sm.Name = "New Name";

sm.gadgetId = 2; // Не доступно

sm.UpdateNameVersion(); // Не доступно

}

В производном классе можно описывать любые допустимые для классов конструкции. Т.е. производный класс состоит из собственных членов и элементов родителя. Иногда бывает нужно изменить поведение методов, унаследованных от базового класса.

Замена функциональности метода базового класса на новую, называется ***переопределением метода***.

**Наследование типов в C#**

Помимо тех типов, которые наследуются через механизм одиночного наследования, все типы в системе типов .NET неявно наследуются от типа [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) или его производного типа. Общие функции [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) доступны любому типу. Неявное наследование от класса [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) делает доступными для всех произвольных классов следующие методы:

* Открытый метод ToString, который преобразует объект вашего класса в строковое представление, возвращает полное имя типа.
* Три метода, которые проверяют равенство двух объектов: открытый метод экземпляра Equals(Object), открытый статический метод Equals(Object, Object) и открытый статический метод ReferenceEquals(Object, Object). По умолчанию эти методы проверяют ссылочное равенство. Это означает, что две переменные, содержащие объекты, должны ссылаться на один и тот же объект, чтобы считаться равными.
* Открытый метод GetHashCode, который вычисляет значение, позволяющее использовать экземпляр типа в хэшированных коллекциях.
* Открытый метод GetType, который возвращает объект [Type](https://learn.microsoft.com/ru-ru/dotnet/api/system.type), представляющий тип вашего класса.
* Защищенный метод [Finalize](https://learn.microsoft.com/ru-ru/dotnet/api/system.object.finalize), который должен освобождать неуправляемые ресурсы перед тем, как сборщик мусора освободит память объекта.
* Защищенный метод [MemberwiseClone](https://learn.microsoft.com/ru-ru/dotnet/api/system.object.memberwiseclone), который создает неполную копию текущего объекта.

В следующей таблице перечислены категории типов, которые можно создавать на языке C#, и указаны типы, от которых они неявно наследуют. Каждый из базовых типов предоставляет всем типам, которые неявно наследуют от него, разные наборы членов.

| ТАБЛИЦА 2 | |
| --- | --- |
| Категория типа | Неявно наследует от |
| class | [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) |
| struct | [ValueType](https://docs.microsoft.com/ru-ru/dotnet/api/system.valuetype), [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) |
| enum | [Enum](https://docs.microsoft.com/ru-ru/dotnet/api/system.enum), [ValueType](https://docs.microsoft.com/ru-ru/dotnet/api/system.valuetype), [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) |
| delegate | [MulticastDelegate](https://docs.microsoft.com/ru-ru/dotnet/api/system.multicastdelegate), [Delegate](https://docs.microsoft.com/ru-ru/dotnet/api/system.delegate), [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object) |

Пример использования методов класса Object

class Person

{

private string name

public string Name { get; set; }

public Person(string name)

{

this.name = name;

}

public override string ToString()

{

return name;

}

public override int GetHashCode()

{

return name.GetHashCode();

}

public override bool Equals(object obj)

{

if (obj.GetType() != this.GetType()) return false;

Person person = (Person)obj;

return (this.Name == person.Name);

}

public static Person[] persons =

{

new Person("Maxim"), new Person("Vladislav"),

new Person("Irina")

};

**Задание на практическую работу №3**

В задании 3 варианта реализации базового задания. Варианты определяются следующим образом:

|  |  |
| --- | --- |
| Вариант из предыдущего задания | Вариант задания №3 и последующих |
| 0, 3, 6 | 1 |
| 1, 4. 7 | 2 |
| 2, 5 | 3 |

## Задание на практическую работу №3 (8 баллов)

**Требования к программе, общие для всех вариантов**

Определить класс **Person**, который имеет

* закрытое поле типа string, в котором хранится имя;
* закрытое поле типа string, в котором хранится фамилия;
* закрытое поле типа System.DateTime для даты рождения.

В классе **Person** определить конструкторы:

* конструктор c тремя параметрами типа string, string, DateTime для инициализации всех полей класса;
* конструктор без параметров, инициализирующий все поля класса некоторыми значениями по умолчанию.

В классе **Person** определить свойства c методами get и set:

* свойство типа string для доступа к полю с именем;
* свойство типа string для доступа к полю с фамилией;
* свойство типа DateTime для доступа к полю с датой рождения;
* свойство типа int c методами get и set для получения информации(get) и изменения (set) года рождения в закрытом поле типа DateTime, в котором хранится дата рождения.

В классе **Person** определить

* перегруженную(override) версию виртуального метода string ToString() для формирования строки со значениями всех полей класса;
* виртуальный метод string ToShortString(), который возвращает строку, содержащую только имя и фамилию.

Cравнить время, необходимое для выполнения операций с элементами одномерного, двумерного прямоугольного и двумерного ступенчатого массивов с одинаковым числом элементов.

Для этого в методе **Main()** создать

* одномерный массив;
* двумерный прямоугольный массив;
* двумерный ступенчатый массив.

Тип элементов массивов зависит от варианта лабораторной работы. Массивы должны иметь одинаковое число элементов. Если число строк в двумерном прямоугольном массиве равно nrow, а число столбцов ncolumn, то одномерный массив должен содержать nrow\*ncolumn элементов, в двумерном ступенчатом массиве общее число элементов также должно быть равно nrow\*ncolumn.

Значения nrow и ncolumn вводятся в процессе работы приложения в виде одной строки с разделителями. В приглашении, которое получает пользователь, должна быть информация о том, какие символы можно использовать как разделители, число разделителей должно быть больше 1. С помощью метода Split класса System.String приложение разбирает введенную пользователем текстовую строку с информацией о числе строк и числе столбцов двумерного массива и присваивает значения переменным, которые содержат значения nrow и ncolumn. В первой лабораторной работе не требуется обрабатывать ошибки ввода, предполагается, что пользователь правильно ввел данные.

Приложение распределяет память для всех массивов и инициализирует элементы массивов. Для инициализации элементов можно использовать конструктор без параметров.

Для всех элементов массивов выполняется одна и та же операция, например, присваивается значение одному из свойств, определенных для элементов массива. В практической работе требуется сравнить время выполнения этой операции для одномерного, двумерного прямоугольного и двумерного ступенчатого массивов с одинаковым числом элементов.

Для измерения времени выполнения операций можно использовать свойство Environment.TickCount. Cтатическое свойство TickCount класса Environment имеет тип int, использует информацию системного таймера и содержит время в миллисекундах, которое прошло с момента перезагрузки компьютера.

Чтобы получить время выполнения некоторого блока кода, необходимо вызвать Environment.TickCount непосредственно перед блоком и сразу же после последнего оператора блока и взять разность значений.

В блоке кода, для которого измеряется время, не должно быть операций распределения памяти для массивов, инициализации элементов массивов и операций вывода данных на консоль. Блоки кода должны содержать только операции с элементами массива.

Вычисленные значения времени выполнения операций для трех типов массивов, а также число строк nrow и столбцов ncolumn выводятся на консоль. Вывод должен быть подписан, т.е. вывод должен содержать информацию о том, какому типу массива отвечает выведенное значение.

## Вариант 1. Требования к программе

Определить тип **Education** - перечисление(enum) со значениями Specialist, Вachelor, SecondEducation.

Определить класс **Exam**, который имеет три открытых автореализуемых свойства, доступных для чтения и записи:

* свойство типа string, в котором хранится название предмета;
* свойство типа int, в котором хранится оценка;
* свойство типа System.DateTime для даты экзамена.

В классе **Exam** определить:

* конструктор с параметрами типа string, int и DateTime для инициализации всех свойств класса;
* конструктор без параметров, инициализирующий все свойства класса некоторыми значениями по умолчанию;
* перегруженную(override) версию виртуального метода string ToString() для формирования строки со значениями всех свойств класса.

Определить класс **Student**, который имеет

* закрытое поле типа Person, в котором хранятся данные студента;
* закрытое поле типа Education для информации о форме обучения;
* закрытое поле типа int для номера группы;
* закрытое поле типа Exam [] для информации об экзаменах, которые сдал студент.

В классе **Student** определить конструкторы:

* конструктор c параметрами типа Person, Education, int для инициализации соответствующих полей класса;
* конструктор без параметров, инициализирующий поля класса значениями по умолчанию.

В классе **Student** определить свойства c методами get и set:

* свойство типа Person для доступа к полю с данными студента;
* свойство типа Education для доступа к полю с формой обучения;
* свойство типа int для доступа к полю с номером группы;
* свойство типа Exam [] для доступа к полю со списком экзаменов.

В классе **Student** определить

* свойство типа double ( только с методом get), в котором вычисляется средний балл как среднее значение оценок в списке сданных экзаменов;
* индексатор булевского типа (только с методом get) с одним параметром типа Education; значение индексатора равно true, если значение поля с формой обучения студента совпадает со значением индекса, и false в противном случае;
* метод void AddExams ( params Exam [] ) для добавления элементов в список экзаменов;
* перегруженную версию виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список экзаменов;
* виртуальный метод string ToShortString(), который формирует строку со значениями всех полей класса без списка экзаменов, но со значением среднего балла.

**В методе Main()**

5. Создать один объект типа Student, преобразовать данные в текстовый вид с помощью метода ToShortString() и вывести данные.

6. Вывести значения индексатора для значений индекса Education.Specialist, Education.Bachelor и Education.SecondEducation.

7. Присвоить значения всем определенным в типе Student свойствам, преобразовать данные в текстовый вид с помощью метода ToString() и вывести данные.

8. C помощью метода AddExams( params Exam[] ) добавить элементы в список экзаменов и вывести данные объекта Student, используя метод ToString().

9. Сравнить время выполнения операций с элементами одномерного, двумерного прямоугольного и двумерного ступенчатого массивов с одинаковым числом элементов типа Exam.

## Вариант 2. Требования к программе

Определить тип **Frequency** - перечисление(enum) со значениями Weekly, Monthly, Yearly.

Определить класс **Article**, который имеет три открытых автореализуемых свойства, доступных для чтения и записи:

* свойство типа Person, в котором хранятся данные автора статьи;
* свойство типа string для названия статьи;
* свойство типа double для рейтинга статьи.

В классе **Article** определить:

* конструктор c параметрами типа Person, string, double для инициализации всех свойств класса;
* конструктор без параметров, инициализирующий все свойства класса некоторыми значениями по умолчанию
* Перегруженную (override) версию виртуального метода string ToString() для формирования строки со значениями всех свойств класса.

Определить класс **Magazine**, который имеет

* закрытое поле типа string c названием журнала;
* закрытое поле типа Frequency с информацией о периодичности выхода журнала;
* закрытое поле типа DateTime c датой выхода журнала;
* закрытое поле типа int с тиражом журнала;
* закрытое поле типа Article[] со списком статей в журнале.

В классе **Magazine** определить конструкторы:

* конструктор с параметрами типа string, Frequency, DateTime, int для инициализации соответствующих полей класса;
* конструктор без параметров, инициализирующий поля класса значениями по умолчанию.

В классе **Magazine** определить свойства c методами get и set:

* свойство типа string для доступа к полю с названием журнала;
* свойство типа Frequency для доступа к полю с информацией о периодичности выхода журнала;
* свойство типа DateTime для доступа к полю c датой выхода журнала;
* свойство типа int для доступа к полю с тиражом журнала;
* свойство типа Article[] для доступа к полю со списком статей.

В классе **Magazine** определить

* свойство типа double ( только с методом get), в котором вычисляется среднее значение рейтинга в списке статей;
* индексатор булевского типа (только с методом get) с одним параметром типа Frequency; значение индексатора равно true, если значение поля типа Frequency совпадает со значением индекса, и false в противном случае;
* метод void AddArticles (params Article[]) для добавления элементов в список статей в журнале;
* перегруженную версию виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список статей;
* виртуальный метод string ToShortString(), который формирует строку со значениями всех полей класса без списка статей, но со значением среднего рейтинга статей.

В методе **Main**()

1. Создать один объект типа Magazine, преобразовать данные в текстовый вид с помощью метода ToShortString() и вывести данные.

2. Вывести значения индексатора для значений индекса Frequency.Weekly, Frequency.Monthly и Frequency.Yearly.

3. Присвоить значения всем определенным в типе Magazine свойствам, преобразовать данные в текстовый вид с помощью метода ToString() и вывести данные.

4. C помощью метода AddArticles( params Article[] ) добавить элементы в список статей и вывести данные объекта Magazine, используя метод ToString().

5. Сравнить время выполнения операций с элементами одномерного, двумерного прямоугольного и двумерного ступенчатого массивов с одинаковым числом элементов типа Article.

## Вариант 3. Требования к программе

Определить тип **TimeFrame** - перечисление(enum) со значениями Year, TwoYears, Long.

Определить класс **Paper**, который имеет три открытых автореализуемых свойства, доступных для чтения и записи:

* свойство типа string, в котором хранится название публикации;
* свойство типа Person для автора публикации;
* свойство типа DateTime c датой публикации.

В классе **Paper** определить

* конструктор c параметрами типа string, Person, DateTime для инициализации всех свойств класса;
* конструктор без параметров, инициализирующий все свойства класса некоторыми значениями по умолчанию;
* перегруженную(override) версию виртуального метода string ToString() для формирования строки со значениями всех полей класса.

Определить класс **ResearchTeam**, который имеет

* закрытое поле типа string c названием темы исследований;
* закрытое поле типа string с названием организации;
* закрытое поле типа int – регистрационный номер;
* закрытое поле типа TimeFrame для информации о продолжительности исследований;
* закрытое поле типа Paper[], в котором хранится список публикаций.

В классе **ResearchTeam** определить конструкторы:

* конструктор c параметрами типа string, string, int, TimeFrame для инициализации соответствующих полей класса;
* конструктор без параметров, инициализирующий поля класса значениями по умолчанию.

В классе **ResearchTeam** определить свойства c методами get и set:

* свойство типа string для доступа к полю с названием темы исследований;
* свойство типа string для доступа к полю с названием организации;
* свойство типа int для доступа к полю с номером регистрации;
* свойство типа TimeFrame для доступа к полю с продолжительностью исследований;
* свойство типа Paper[] для доступа к полю со списком публикаций по теме исследований.

В классе **ResearchTeam** определить

* свойство типа Paper (только с методом get), которое возвращает ссылку на публикацию с самой поздней датой выхода; если список публикаций пустой, свойство возвращает значение null;
* индексатор булевского типа (только с методом get) с одним параметром типа TimeFrame; значение индексатора равно true, если значение поля с информацией о продолжительности исследований совпадает со значением индекса, и false в противном случае;
* метод void AddPapers ( params Paper[] ) для добавления элементов в список публикаций;
* перегруженную версию виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список публикаций;
* виртуальный метод string ToShortString(), который формирует строку со значениями всех полей класса без списка публикаций.

В методе **Main**()

1. Создать один объект типа ResearchTeam, преобразовать данные в текстовый вид с помощью метода ToShortString() и вывести данные.

2. Вывести значения индексатора для значений индекса TimeFrame.Year, TimeFrame.TwoYears, TimeFrame.Long.

3. Присвоить значения всем определенным в типе ResearchTeam свойствам, преобразовать данные в текстовый вид с помощью метода ToString() и вывести данные.

4. С помощью метода AddPapers (params Paper []) добавить элементы в список публикаций и вывести данные объекта ResearchTeam.

5. Вывести значение свойства, которое возвращает ссылку на публикацию с самой поздней датой выхода;

6. Сравнить время выполнения операций с элементами одномерного, двумерного прямоугольного и двумерного ступенчатого массивов с одинаковым числом элементов типа Paper.

# Практическая работа №4. Абстракция, полиморфизм

Цель: отличие абстракции от виртуальности, отличие перегрузки от переопределения.

**Полиморфизм** – способность функции обрабатывать данные разных типов, имеющих одинаковый интерфейс (базовый класс).

В производном классе можно описывать любые допустимые для классов конструкции.

Т.е. производный класс состоит из собственных членов и элементов родителя. Иногда бывает нужно изменить поведение методов, унаследованных от базового класса. Замена функциональности метода базового класса на новую, называется переопределением метода.

**Виртуальные методы**

Нельзя просто взять и переопределить любой унаследованный метод.

Методы, которые разрешается переопределять, должны быть помечены в базовом классе ключевым словом virual.

Такие методы называются виртуальными методами. Если метод виртуальный, это не значит, что он должен быть переопределён. Это значит, что он может быть переопределён.

Чтобы переопределить виртуальный метод, нужно написать в производном классе метод такой же сигнатуры (с таким же именем, спецификатором доступа, типом возвращаемого значения и набором параметров) и пометить его ключевым словом override. Даже если метод переопределён, всё равно остаётся возможность обратиться к первоначальной версии метода.

С помощью ключевого слова base можно обращаться к содержимому базового класса. Получается, что внутри класса есть два специальных ключевых слова this и base. Слово this используется для доступа к элементам текущего класса (включая унаследованные элементы). Слово base используется для доступа к оригинальным элементам базового класса.

**Абстрактный класс** – незавершенный класс, который используется только для наследования. Для такого класса невозможно создать объект.

Обычно, абстрактные классы используются в качестве начального компонента иерархии классов.

Или просто, когда класс описывает какую-то абстрактную сущность, от которой всегда должны наследоваться другие классы. Абстрактный класс помечается ключевым словом abstract.

Только в абстрактном классе могут содержаться абстрактные члены класса.

Абстрактными могут быть методы, свойства, индексаторы и события.

Абстрактный член класса не имеет реализации, и должен обязательно быть реализован в производных классах.

Абстрактные члены класса также помечаются ключевым словом abstract.

В производных классах реализация абстрактных членов класса должна быть помечена ключевым словом override. Свойства, помеченные как абстрактные, не должны иметь реализации, т.е. у них не должно быть кода аксессоров get и set. Абстрактные методы не должны иметь тела функции, только сигнатуру вызова.

Ключевые слова abstract и static несовместимы. Собственно, как и virtual со static.

Статические члены класса в принципе не могут быть ни абстрактными, ни виртуальными, ни какими бы то ни было ещё - только статическими.

## Задание к практической работе №4 (5 баллов).

Модифицируйте программу из практической № 2, построив на его основе иерархию (через наследование), как минимум из 5 классов. Корневой класс иерархии должен быть абстрактным, а хотя бы один класс из потомков – бесплодным. У каждого из классов должно быть хотя бы одно собственное свойство и метод. Также должны быть продемонстрированы виртуальные и переопределённые методы.

Вариант задания остаётся прежним.

|  |  |
| --- | --- |
| Вар. | Задание |
| 0 | Примеры классов: кнопка, элемент управления, текстовое поле, чекбокс, круглая кнопка, кнопка с изображением. |
| 1 | Примеры классов: вертолёт, Ми-6, самолёт, летательный аппарат, бомбардировщик, боевой вертолёт. |
| 2 | Примеры классов: четырёхугольная пирамида, объект сцены, сфера, куб, трёхмерная фигура. |
| 3 | Примеры классов: герой, воин, лекарь, персонаж, лучник, горожанин. |
| 4 | Примеры классов: автомобиль, снегоход, микроавтобус, внедорожник, транспортное средство, мотоцикл. |
| 5 | Примеры классов: работник, персона, менеджер, работник службы поддержки, работник бухгалтерии, директор. |
| 6 | Примеры классов: пользователь, администратор, редактор, пользователь с платной подпиской, тестировщик, принципал (объект, содержащий идентификационные сведения). |
| 7 | Примеры классов: банковский перевод, обмен валют, операция, платёж, оплата коммунальных услуг. |

# Практическая работа №5. Интерфейсы

Цель: знакомство с интерфейсами, перегрузка операций, полиморфизм, исключения, наследование

**Интерфейсы**

К абстрактным классам близки Интерфейсы.

Интерфейс (или Протокол) – набор абстрактных членов (обычно методов), которые должны быть обязательно реализованы классами с этим интерфейсом.

Интерфейсы описываются с помощью ключевого слова interface. Интерфейсы, это отдельные элементы в программировании, наподобие классов, структур, перечислений. Имена интерфейсов должны начинаться на букву I. Как мы уже знаем, любой класс может наследоваться от другого класса.

Помимо этого:

любой класс может реализовывать любое количество интерфейсов. Для обозначения реализации интерфейса используется такая же нотация, как и при наследовании. Единственное, что имя родительского класса, если он есть, должно всегда идти первым в списке.

class SomeClass1 : IInterface1

{

// ...

}

class SomeClass2 : ParentClass, IInterface1

{

// ...

}

class SomeClass3 : ParentClass, IInterface1, IInterface2, IInterface3

{

// ...

}

Интерфейсы состоят из абстрактных методов (или других членов класса).

В отличие от абстрактных классов, в интерфейсе не может присутствовать ничего другого, т.е. никакой реализации (до версии C#8.0). Абстрактные члены в абстрактном классе могут иметь различные спецификаторы доступа, а в интерфейсе элементы всегда public.

Класс может наследовать лишь 1 абстрактный класс, а интерфейсов иметь неограниченное количество. Но главное различие – идейное.

Абстрактные классы используют, когда необходимо описать какую-то неопределённую сущность, понятие, которое будет раскрываться в дальнейшем, через производные классы.

А интерфейсы описывают способность делать что-то, наличие какой-либо функциональности.

Например, интерфейс ICallable – описывает способность совершать и отвечать на звонки. Будет реализовываться классами Telephone, Smartphone, Tablet и WalkieTalkie.

Поэтому имена интерфейсов принято заканчивать суффиксом -able. Методы в интерфейсах описываются без всяких дополнительных ключевых слов, и даже без спецификаторов доступа.

Вот при реализации классом, все методы из интерфейса уже должны быть со спецификатором доступа public.

**Наследование интерфейсов**

Интерфейсы, также, как и классы, могут наследовать другие интерфейсы.

Но тут нет ничего особенного. Если у класса в интерфейсах встретятся методы с одинаковой сигнатурой, одна реализация подойдет для обоих интерфейсов.

**Стерильные классы**

Или запечатанные классы – это классы, от которых запрещено наследоваться, от такого класса невозможно создать производный класс.

Запечатанными обычно назначаются классы в распространяемых библиотеках, которые имеют логически завершенную структуру и функциональность, и расширение которых больше не предполагается.

Для того чтобы отметить класс как запечатанный, используется ключевое слово sealed.

Методы тоже можно запечатывать. Но не все, а только переопределённые виртуальные. Таким образом, они защищаются от последующего переопределения. Для этого метод помечается тем же ключевым словом sealed.

**Сокрытие методов**

Помимо переопределения методов, есть ещё и сокрытие методов.

**Сокрытие** – это создание в производном классе метода, одинакового по сигнатуре с методом в базовом классе. Метод в производном классе перекрывает метод в базовом классе. Т.е. это написание метода с таким же именем и параметрами, как в базовом классе.

И за счёт совпадения имён будет виден только один из них, а именно скрывающий метод в производном классе. Для сокрытия метода, скрывающий метод должен быть помечен ключевым словом new.

Главным образом сокрытие нужно тогда, когда вам в классе обязательно нужен метод с каким-то названием, а метод с таким названием уже есть в базовом классе, и он не виртуальный.

Из-за того, что он не виртуальный, нельзя выполнить переопределение этого метода. И тогда единственным способом описать метод с таким же именем остаётся механизм сокрытия.

class Dog

{

public string Name { get; set; }

public string Breed { get; set; }

public virtual void Walk()

{

Console.WriteLine("The {0} walks!", Name);

Random randGen = new Random();

int rndNumber = randGen.Next(4);

if (rndNumber == 3)

{

this.Pee();

}

}

private int Pee()

{

Console.WriteLine("The {0} has urinated..", Name);

Random randGen = new Random();

int strenth = randGen.Next(1, 11);

return strenth;

}

// Переопределение

public override string ToString()

{

return $"Dog named {Name}, with owner {Owner}, {Breed}, {Age} years";

}

// Новый метод

public int GetStamina()

{

int stamina = 20 \* (8 - Math.Abs(Age - 8)) / 8;

Console.WriteLine("The dog stamina: {0}/100", stamina);

return stamina;

}

class PoliceDog : Dog

{

public int Rank { get; set; }

public string Specialization { get; set; }

public PoliceDog(string name, int age, string breed, string owner, int rank, string spec)

{

Name = name;

Age = age;

Rank = rank;

Specialization = spec;

}

public int UpRank()

{

++Rank;

return Rank;

}

// Переопределённый метод

public override string ToString()

{

return "Police " + base.ToString() + $". With rank {Rank} and specializing in {Specialization}";

}

// Скрывающий метод

public new int GetStamina()

{

// Изменённая формула - вместо 20 здесь значение 100

int stamina = 100 \* (8 - Math.Abs(Age - 8)) / 8;

Console.WriteLine("The police dog stamina: {0}/100", stamina);

return stamina;

}

**static void Main(string[] args)**

{

// Создаём объект класса Dog

Dog dog1 = new Dog();

dog1.Name = "Mikkey";

dog1.Breed = "American cocker spaniel";

// Создаём объект класса PoliceDog

PoliceDog dog2 = new PoliceDog("Ralf", "Indian pariah dog", 3, "Explosive");

// Вызов ToString класса Dog

dog1.GetStamina();

// Вывод: The dog stamina: 5/100

// Вызов ToString класса PoliceDog, который внутри вызывает ToString класса Dog

dog2.GetStamina();

// Вывод: The police dog stamina: 100/100

} }

При сокрытии в классе остаётся 2 варианта метода: скрытый, унаследованный от базового класса, доступ к которому нельзя получить. И новый, видимый, который перекрывает предыдущий.

Полиморфизмом называется способность одних типов выступать в форме других типов. Или же способность функции обрабатывать данные разных типов.

Наследование - это отношение "является". Любой производный класс "является" сущностью, которую представляет его базовый класс.

Пусть у нас есть класс Fruit. И класс Banana, производный от него. А ещѐ класс Apple, тоже потомок Fruit.

В таком случае, Banana "является" Fruit и Apple "является" Fruit.

Это свойство классов проявляется в коде в виде возможности представить объект какого-то класса в виде объекта базового для него класса.

// Базовый класс Фрукт

class Fruit

{

// Свойство Название

public string Name { get; set; }

}

// Производный от Фрукта класс Банан

class Banana : Fruit

{

// Свойство Длина

public int Length { get; set; }

}

// Производный от Фрукта класс Яблоко

class Apple : Fruit

{

// Свойство Цвет

public string Color { get; set; }

}

static void Main(string[] args)

{

// Создаём экземпляр класса Banana

Banana ban = new Banana();

// Присваиваем объект типа Banana ссылке на тип Fruit

Fruit fruit = ban;

// Создаём экземпляр класса Apple

Apple ap = new Apple();

// Присваиваем объект типа Aplle ссылке на тип Fruit

Fruit fr2 = ap;

// Создаём ещё один объект типа Apple и помещаем его в тип Fruit

Fruit fr3 = new Apple();

} }

**Виды полиморфизма**

Полиморфизм бывает 2 видов:

* параметрический (истинный) полиморфизм;
* ad hoc полиморфизм.

Ad hoc полиморфизм является не совсем настоящим полиморфизмом. Он наблюдается тогда, когда мы вручную прописываем различные варианты поведения для различных типов.

**Ad hoc полиморфизм**

Простейшим случаем Ad hoc полиморфизма можно считать перегрузку функций: когда мы для различных типов параметров прописываем различные варианты методов.

// Класс Шеф-повара

class Chef

{

// Метод Приготовить что-нибудь из банана

public string CookSmth(Banana b)

{

return "Banana cake";

}

// Метод Приготовить что-нибудь из яблока

public string CookSmth(Apple a)

{

return "Charlotte";

}

// Метод Приготовить что-нибудь из киви

public string CookSmth(Kiwi k)

{

return "Smoothie";

} }

Ещё один вариант Ad hoc полиморфизма – когда для всех типов у нас используется один общий метод, но всё равно проверяется исходный тип и для каждого типа выполняются отдельные действия.

// Метод Приготовить что-нибудь из фрукта

public string CookSmth(Fruit fruit)

{

string meal = "nothing";

// Если тип фрукта - банан

if (fruit is Banana)

{

meal = "Banana cake";

}

// Если тип фрукта - яблоко

else if (fruit is Apple)

{

meal = "Charlotte";

}

// Если тип фрукта - киви

else if (fruit is Kiwi)

{

meal = "Smoothie";

}

return meal; }}

static void Main(string[] args)

{

// Создаём шефа

Chef mrChef = new Chef();

// По одному экземпляру каждого класса

Fruit fruit1 = new Fruit();

Banana fruit2 = new Banana();

Apple fruit3 = new Apple();

Kiwi fruit4 = new Kiwi();

// И смотрим варианты вызова метода CookSmth, передавая в него объекты разных типов

string meal = mrChef.CookSmth(fruit1);

Console.WriteLine("First meal: " + meal);

meal = mrChef.CookSmth(fruit4);

Console.WriteLine("And final meal: " + meal);

}

**Коллекции и полиморфизм**

Благодаря полиморфизму мы можем не только создавать методы, которые будут работать с разными типами, как с одним, но и работать с коллекциями из объектов разных классов.

Зная, что объекты классов Banana, Apple и Kiwi могут быть представлены как объекты класса Fruit, мы спокойно можем создать массив фруктов, и поместить в него все наши объекты.

static void Main(string[] args)

{

Chef mrChef = new Chef();

// Объявляем массив фруктов и инициализируем его списком объектов разных классов

Fruit[] fruits = { new Fruit(), new Banana(), new Apple(), new Kiwi() };

// Циклом перебираем фрукты в массиве

foreach (Fruit fruit in fruits)

{

string meal = mrChef.CookSmth(fruit);

Console.WriteLine("Next meal: " + meal);

}

}

**Оператор is** используется для проверки, "является" ли объект указанным типом, т.е. находится ли этот тип в иерархии наследования проверяемого объекта.

Главная особенность именно в том, что он проверяет не только тот тип, которым создавался объект, а все типы в его иерархии наследования (а также реализуемые интерфейсы).

Это значит, что, например, для объекта класса Apple положительными были бы проверки, что он является Apple, Fruit и Object.

**Класс Type**

Существует и другой способ, как узнать какой конкретно тип у объекта, через класс Type.

Type – специальный класс, представляющий собой информацию о каком-либо типе. Для любого типа (класса) можно получить соответствующий ему объект класса Type.

И именно для этого в классе Object есть метод GetType(). А раз он есть в классе Object, он есть любом классе .NET. Этот метод возвращает объект типа для своего объекта.

**Оператор typeof**

Оператор typeof возвращает объект класса Type для заданного типа.

Синтаксис применения у него следующий:

typeof(тип)

Итак, GetType позволяет получить объект Type для объекта, typeof позволяет получить объект Type для типа. Скомбинировав их, мы можем записать проверку на однозначное соответствие объекта какому-либо типу:

объект.GetType() == typeof(тип)

**Пример**:

Banana banan = new Banana();

Console.WriteLine("Operator is:");

Console.WriteLine(" banan is Banana: {0}", banan is Banana); // true

Console.WriteLine(" banan is Kiwi: {0}", banan is Kiwi); // false

Console.WriteLine(" banan is Fruit: {0}", banan is Fruit); // true

Console.WriteLine(" banan is Object: {0}", banan is Object); // true

Console.WriteLine("Class Type:");

Console.WriteLine(" banan.GetType() == typeof(Banana): {0}", banan.GetType() == typeof(Banana)); // true

Console.WriteLine(" banan.GetType() == typeof(Kiwi): {0}", banan.GetType() == typeof(Kiwi)); // false

Console.WriteLine(" banan.GetType() == typeof(Fruit): {0}", banan.GetType() == typeof(Fruit)); // false

Console.WriteLine(" banan.GetType() == typeof(Object): {0}", banan.GetType() == typeof(Object)); // false

**Обратное преобразование**

Иногда бывает нужно восстановить исходный тип объекта.

Например, передав объект класса Banana в метод в виде Fruit, восстановить из этого Fruit исходный тип Banana.

Если попробовать выполнить преобразование через обычный синтаксис приведения типов, в случае, если попадётся не совместимый тип, программа выбросит исключение.

public void SomeMethod(Fruit fruit)

{

// В случае, если сюда передать любой фрукт, кроме банана. Произойдет исключение - преобразование типов невозможно

Banana ban = (Banana)fruit;

Console.WriteLine("Banana length is: " + ban.Length);

}

**Оператор as**

Оператор as выполняет преобразование объекта в тип, находящийся в его иерархии наследования, а если указанный тип не связан с объектом – возвращает нулевую ссылку null. Исключений здесь не происходит.

Синтаксис у него такой:

ожидаемый\_тип новое\_имя = объект as ожидаемый\_тип;

Оператор as подходит только для преобразования типов, связанных наследственными узами.

Banana ban = fruit as Banana;

if (ban != null)

{

Console.WriteLine("Banana length is: " + ban.Length);

}

Вообще, с 7 версии C# оператор as стал немного бесполезным, потому что его функциональность добавили к новой форме оператора is:

// Теперь преобразовывать можно прямо в операторе is

if (fruit is Banana ban)

{

// Если проверка is пройдёт, то здесь будет доступен преобразованный объект ban

Console.WriteLine("Banana length is: " + ban.Length);

}

**Боксинг и анбоксинг**

Но у такого представления в виде типа object есть и свои преимущества. А именно, если говорить о хранении в виде object структур (стандартных типов данных: int, float, bool и т.д.)

Структуры являются типами значений, а это значит, что в методы передаются копии их значений. А Object является классом, т.е. ссылочным типом, и объекты типа object передаются в методы по ссылке.

У этого даже есть специальные названия:

Представление типа значения в виде object – боксинг.

А возвращение типа значения из типа object в объект исходного типа – это анбоксинг.

**Виды полиморфизма**

Ещё одна классификация видов полиморфизма.

По типу применяемых элементов полиморфизм делится на:

* полиморфизм подтипов;
* полиморфизм интерфейсов.

Мы пока рассматриваем полиморфизм подтипов (т.е. на основе наследования). И до этого он у нас был ненастоящим (ad hoc полиморфизм).

Истинный (параметрический) полиморфизм.

Полиморфизм считается истинным, когда в методе описан только один сценарий поведения для любых входных типов. Но, тем не менее, поведение меняется в зависимости от типа входящего объекта.

Достичь этого можно с помощью перегрузки методов.

Описав в базовом классе виртуальный метод, функция, работающая с этим базовым классом, может вызывать этот виртуальный метод. А он, в свою очередь, у каждого производного класса будет переопределён и реализован по-своему.

Т.е. вызываться будет один метод, но он будет делать разные вещи, в зависимости от того, как он переопределён в этом классе.

Пример настоящего полиморфизма

class Fruit

{

public string Name { get; set; }

// Виртуальный метод "Получить блюдо"

public virtual string GetMealFromThis()

{

return "Nothing";

}

}

class Banana : Fruit

{

public int Length { get; set; }

// Этот метод переопределён

// И для банана возвращает такую строку

public override string GetMealFromThis()

{

return "Banana Cake";

}

}

class Apple : Fruit

{

public string Color { get; set; }

// Для яблока вторую

public override string GetMealFromThis()

{

return "Charlotte";

}

}

class Kiwi : Fruit

{

public string Size { get; set; }

// А для киви третью

public override string GetMealFromThis()

{

return "Smoothie";

}

}

class Chef

{

// Метод Приготовления

public string CookSmth(Fruit fruit)

{

// Просто возвращает результат выполнения метода GetMealFromThis

return fruit.GetMealFromThis();

}

}

**static void Main(string[] args)**

{

Chef mrChef = new Chef();

Fruit[] fruits = { new Fruit(), new Banana(), new Apple(), new Kiwi() };

foreach (Fruit fruit in fruits)

{

string meal = mrChef.CookSmth(fruit);

Console.WriteLine("Next meal: " + meal);

}

}

**Переопределение и сокрытие**

При переопределении, как вы увидели, в случае представления объекта в форме базового класса, при вызове виртуального метода вызывается его переопределённая версия из дочернего класса.

При сокрытии же такого не происходит, и полиморфизм не работает. Для объекта базового класса всегда будет выполняться вариант метода, описанный в базовом классе. Подробные примеры можно посмотреть в лекции.

**Полиморфизм интерфейсов**

Так же как и с базовыми классами, объекты, реализующие какой-либо интерфейс, могут быть представлены в виде объекта типа этого интерфейса. Такой тип полиморфизма называется Полиморфизм интерфейсов.

interface IBarkable

{

// Метод Лаять

void Bark(int power);

// Метод Скулить

void Whine();

}

// Класс Собаки, реализующего интерфейс "Лающее"

class Dog : IBarkable

{

public string Name { get; set; }

public string Breed { get; set; }

// реализация методов остается прежней

public virtual void Walk()

{… }

private int Pee()

{… }

public override string ToString()

{… }

// Реализация метода Лаять

public void Bark(int power)

{ switch (power)

{

case 0:

Whine();

break;

case 1:

Console.WriteLine(" - Woof...");

break; }}

// Реализация метода Скулить

public void Whine()

{

Console.WriteLine(" - Whiiee..");

}

class PoliceDog : Dog

{

// ...

}

// Класс Кота

class Cat

{

private Random generator;

// Конструктор

public Cat()

{

// Для инициализации генератора случайных чисел

generator = new Random();

}

// Метод Мяукнуть

protected string Meow()

{

string word = " - Meeeooowwww!";

Console.WriteLine(word);

return word;

}

// Метод Убежать от кого-то лающего. В качестве параметра - объект интерфейса

public void RunAwayFrom(IBarkable barkable)

{ // Гавкаем и мяукаем друг на друга

int rnd, iter = 1;

do

{

Meow();

rnd = generator.Next(2);

barkable.Bark(iter);

++iter;

}

while (rnd != 0);

// И убегаем

Console.WriteLine("The cat ran away and climbed a tree.\n");

}

static void Main(string[] args)

{

Cat cat = new Cat();

Dog dog1 = new Dog();

cat.RunAwayFrom(dog1);

PoliceDog dog2 = new PoliceDog();

cat.RunAwayFrom(dog2);

// Создаём экземпляр собаки и присваеваем объекту интерфейса

IBarkable barkable = new Dog();

cat.RunAwayFrom(barkable);

}

**Явная реализация интерфейсов**

Помимо обычной реализации членов интерфейса существуем механизм явной реализации. Явно реализованный член интерфейса будет доступен в объекте только тогда, когда объект будет представлен в форме этого интерфейса.

С помощью этого можно "прятать" методы внутри классов, а также управлять поведением членов класса в зависимости от формы их представления. Механизм явной и неявной реализации интерфейсов подробно рассмотрен в лекции.

Для полиморфизма, как правило, используют именно интерфейсы. Наследование имеет значительное ограничение из-за невозможности иметь нескольких родителей, поэтому на практике его применяют в основном только для переиспользования кода.

Полиморфизм интерфейсов даёт нам необычайную возможность строить программу на основе интерфейсов без фиксированной реализации.

Такая способность открывает возможность легкой поддержки и модификации любых частей кода, расширяемость и адаптируемость, простоту тестирования и дополнительных интеграций.

Почти во всех случаях грамотная архитектура ООП приложения должна опираться именно на Интерфейсы. Любые зависимости следует представлять в виде интерфейсов, всё взаимодействие программных компонентов должно вестись через интерфейсы.

Для построения правильной архитектуры ПО нужно мыслить интерфейсами, а не классами: «Интерфейсы - для людей, классы - для животных».

Подробно о стандартных интерфейсах смотреть в лекции «Стандартные интерфейсы»

## Задание на практическую работу №5 (8 баллов)

В классе **Person** и в классах, дополнительно указанных в вариантах, надо

1. переопределить override) виртуальный метод bool Equals (object obj);
2. определить операции == и != ;
3. переопределить виртуальный метод int GetHashCode();

Реализация виртуального метода bool Equals (object obj) в классе System.Object определяет равенство объектов как равенство ссылок на объекты. Некоторые классы из базовой библиотеки BCL переопределяют метод Equals(). В классе System.String этот метод переопределен так, что равными считаются строки, которые совпадают посимвольно. Реализация метода Equals() в структурном типе DateTime равенство объектов DateTime определяет как равенство значений.

В работе требуется переопределить метод Equals так, чтобы объекты считались равными, если равны все данные объектов. Для класса **Person** это означает, что равны даты рождения и посимвольно совпадают строки с именем и фамилией.

Определение операций == и != должно быть согласовано с переопределенным методом Equals, т.е. критерии, по которым проверяется равенство объектов в методе Equals, должны использоваться и при проверке равенства объектов в операциях == и !=.

Переопределение виртуального метода int GetHashCode() также должно быть согласовано с операциями == и !=. Виртуальный метод GetHashCode() используется некоторыми классами базовой библиотеки, например, коллекциями-словарями. Классы базовой библиотеки, вызывающие метод GetHashCode() из пользовательского типа, предполагают, что равным объектам отвечают равные значения хэш-кодов. Поэтому в случае, когда под равенством объектов понимается совпадение данных (а не ссылок), реализация метода GetHashCode() должна для объектов с совпадающими данными возвращать равные значения хэш-кодов.

В классах, указанных в вариантах лабораторной работы, требуется определить метод object DeepCopy() для создания полной копии объекта. Определенные в некоторых классах базовой библиотеки методы Clone() и Copy() создают ограниченную (shallow) копию объекта – при копировании объекта копии создаются только для полей структурных типов, для полей ссылочных типов копируются только ссылки. В результате в ограниченной копии объекта поля-ссылки указывают на те же объекты, что и в исходном объекте.

Метод DeepCopy() должен создать полные копии всех объектов, ссылки на которые содержат поля типа. После создания полная копия не зависит от исходного объекта - изменение любого поля или свойства исходного объекта не должно приводить к изменению копии.

При реализации метода DeepCopy() в классе, который имеет поле типа System.Collections.ArrayList, следует иметь в виду, что определенные в классе ArrayList конструктор ArrayList(ICollection) и метод Clone() при создании копии коллекции, состоящей из элементов ссылочных типов, копируют только ссылки.

Метод DeepCopy() должен создать как копии элементов коллекции ArrayList, так и полные копии объектов, на которые ссылаются элементы коллекции. Для типов, содержащих коллекции, реализация метода DeepCopy() упрощается, если в типах элементов коллекций также определить метод DeepCopy().

## Вариант 1. Требования к программе

Определить интерфейс

**interface IDateAndCopy**

**{ object DeepCopy();**

**DateTime Date { get; set; }**

**}**

Определить новые версии классов **Exam**, **Person** и **Student**. В классы **Exam**, **Person** и **Student** добавить реализацию интерфейса IDateAndCopy. Новую версию класса **Student** определить как класс, производный от класса **Person**.

Все поля новой версии класса **Person** определить с доступом protected, сохранить все свойства, определенные в первой версии класса.

В новой версии класса **Person** дополнительно

* переопределить метод virtial bool Equals (object obj) и определить операции == и != так, чтобы равенство объектов типа Person трактовалось как совпадение всех данных объектов, а не ссылок на объекты Person;
* переопределить виртуальный метод int GetHashCode();
* определить виртуальный метод object DeepCopy();
* реализовать интерфейс IDateAndCopy.
* Определить класс **Test**, который имеет два открытых автореализуемых свойства, доступных для чтения и записи:
* свойство типа string, в котором хранится название предмета;
* свойство типа bool для информации о том, сдан зачет или нет.

В классе **Test** определить:

* конструктор c параметрами типа string и bool для инициализации свойств класса;
* конструктор без параметров, инициализирующий все свойства класса некоторыми значениями по умолчанию;
* перегруженную(override) версию виртуального метода string ToString() для формирования строки со значениями всех свойств класса.

Класс **Student** определить как производный от класса **Person**.

Новая версия класса **Student** имеет следующие поля:

* закрытое поле типа Education для информации о форме обучения;
* закрытое поле типа int для номера группы;
* закрытое поле типа System.Collections.ArrayList, в котором хранится список зачетов (объекты типа Test);
* закрытое поле типа System.Collections.ArrayList для списка экзаменов (объекты типа **Exam**).

Код следующих конструкторов, методов и свойств из старой версии класса **Student** необходимо изменить с учетом того, что часть полей класса перемещена в базовый класс **Person**, и в новой версии класса **Student** список экзаменов хранится в коллекции System.Collections.ArrayList:

* конструктор c параметрами типа Person, Education, int для инициализации соответствующих полей класса;
* конструктор без параметров для инициализации по умолчанию;
* свойство типа Person; метод get свойства возвращает объект типа Person, данные которого совпадают с данными подобъекта базового класса, метод set присваивает значения полям из подобъекта базового класса;
* свойство типа double ( только с методом get), в котором вычисляется средний балл как среднее значение оценок в списке сданных экзаменов;
* свойство типа System.Collections.ArrayList с методами get и set для доступа к полю со списком экзаменов;
* метод void AddExams ( params Exam[] ) для добавления элементов в список экзаменов;
* перегруженная версия виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список зачетов и экзаменов;
* виртуальный метод string ToShortString(), который формирует строку со значениями всех полей класса без списка зачетов и экзаменов, но со значением среднего балла.

Дополнительно в новой версии класса **Student**

* определить перегруженную версию виртуального метода object DeepCopy();
* реализовать интерфейс IDateAndCopy;
* определить свойство типа int с методами get и set для доступа к полю с номером группы. В методе set бросить исключение, если присваиваемое значение меньше или равно 100 или больше 599. При создании объекта-исключения использовать один из определенных в библиотеке CLR классов-исключений, инициализировать объект-исключение с помощью конструктора с параметром типа string, в сообщении передать информацию о допустимых границах для значения свойства.

В новой версии класса **Student** определить

* итератор для последовательного перебора всех элементов (объектов типа object) из списков зачетов и экзаменов (объединение);
* итератор c параметром для перебора экзаменов (объектов типа Exam) с оценкой больше заданного значения.

В методе **Main()**

1. Создать два объекта типа Person с совпадающими данными и проверить, что ссылки на объекты не равны, а объекты равны, вывести значения хэш-кодов для объектов.

2. Создать объект типа Student, добавить элементы в список экзаменов и зачетов, вывести данные объекта Student.

3. Вывести значение свойства типа Person для объекта типа Student.

4. С помощью метода DeepCopy() создать полную копию объекта Student. Изменить данные в исходном объекте Student и вывести копию и исходный объект, полная копия исходного объекта должна остаться без изменений.

5. В блоке try/catch присвоить свойству с номером группы некорректное значение, в обработчике исключения вывести сообщение, переданное через объект-исключение.

6. С помощью оператора foreach для итератора, определенного в классе Student, вывести список всех зачетов и экзаменов.

7. С помощью оператора foreach для итератора с параметром, определенного в классе Student, вывести список всех экзаменов с оценкой выше 3.

**Дополнительное задание: (5 баллов)**

В классе **Student**

* реализовать интерфейс System.Collections.IEnumerable для перебора названий всех предметов (объектов типа string), которые есть как в списке зачетов, так и в списке экзаменов (пересечение). Для этого определить вспомогательный класс StudentEnumerator, реализующий интерфейс System.Collections.IEnumerator.
* определить итератор для перебора сданных зачетов и экзаменов (объектов типа object), для этого определить метод, содержащий блок итератора и использующий оператор yield; сданный экзамен – экзамен с оценкой больше 2;
* определить итератор для перебора всех сданных зачетов (объектов типа Test), для которых сдан и экзамен, для этого определить метод, содержащий блок итератора и использующий оператор yield.

В методе **Main()**

8. С помощью оператора foreach для объекта типа Student вывести список предметов, которые есть как в списке зачетов, так и в списке экзаменов.

9. С помощью оператора foreach для итератора, определенного в классе Student, вывести список всех сданных зачетов и сданных экзаменов.

10. С помощью оператора foreach для итератора, определенного в классе Student, вывести список сданных зачетов, для которых сдан и экзамен.

## Вариант 2. Требования к программе

Определить интерфейс

**interface IRateAndCopy**

**{ double Rating { get;}**

**object DeepCopy();**

**}**

Определить новые версии классов **Person**, **Article** и **Magazine**. Класс **Magazine** определить как производный от класса **Edition**. В классы **Article** и **Magazine** добавить реализацию интерфейса IRateAndCopy.

В новой версии класса **Person** дополнительно

* переопределить метод virtial bool Equals (object obj) и определить операции == и != так, чтобы равенство объектов типа Person трактовалось как совпадение всех данных объектов, а не ссылок на объекты Person;
* переопределить виртуальный метод int GetHashCode();
* определить виртуальный метод object DeepCopy().

В новой версии класса **Article** дополнительно

* определить виртуальный метод object DeepCopy();
* реализовать интерфейс IRateAndCopy.

Определить класс **Edition**. Класс **Edition** имеет

* защищенное(protected) поле типа string c названием издания;
* защищенное поле типа DateTime c датой выхода издания;
* защищенное поле типа int с тиражом издания;

В классе **Edition** определить:

* конструктор с параметрами типа string, DateTime, int для инициализации соответствующих полей класса;
* конструктор без параметров для инициализации по умолчанию;
* свойства c методами get и set для доступа к полям типа;
* виртуальный метод object DeepCopy();
* свойство типа int с методами get и set для доступа к полю с тиражом издания; в методе set свойства бросить исключение, если присваиваемое значение отрицательно. При создании объекта-исключения использовать один из определенных в библиотеке CLR классов-исключений, инициализировать объект-исключение с помощью конструктора с параметром типа string, в сообщении передать информацию о допустимых значениях свойства.

В классе **Edition** переопределить (override):

* виртуальный метод virtial bool Equals (object obj) и определить операции == и != так, чтобы равенство объектов типа Edition трактовалось как совпадение всех данных объектов, а не ссылок на объекты Edition;
* виртуальный метод int GetHashCode();
* перегруженную версию виртуального метода string ToString() для формирования строки со значениями всех полей класса.

Новая версия класса **Magazine** имеет базовый класс **Edition** и следующие поля:

* закрытое поле типа Frequency с информацией о периодичности выхода журнала;
* закрытое поле типа System.Collections.ArrayList со списком редакторов журнала (объектов типа **Person**).
* закрытое поле типа System.Collections.ArrayList, в котором хранится список статей в журнале (объектов типа **Article**).

Код следующих конструкторов, методов и свойств из старой версии класса **Magazine** необходимо изменить с учетом того, что часть полей класса перемещена в базовый класс **Edition**, и в новой версии класса **Magazine** для списка статей используется тип System.Collections.ArrayList:

конструктор с параметрами типа string, Frequency, DateTime, int для инициализации соответствующих полей класса;

* конструктор без параметров для инициализации по умолчанию;
* свойство типа double (только с методом get), в котором вычисляется среднее значение рейтинга статей в журнале;
* свойство типа System.Collections.ArrayList для доступа к полю со списком статей в журнале;
* метод void AddArticles (params Article[]) для добавления элементов в список статей в журнале;
* перегруженная версия виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список статей и список редакторов;
* виртуальный метод string ToShortString(), который формирует строку со значениями всех полей класса без списка статей и списка редакторов, но со значением среднего рейтинга статей в журнале.

Дополнительно в новой версии класса **Magazine** реализовать

* свойство типа System.Collections.ArrayList для доступа к списку редакторов журнала;
* метод void AddEditors (params Person[]) для добавления элементов в список редакторов;
* перегруженную (override) версию виртуального метода object DeepCopy();
* интерфейс IRateAndCopy;
* свойство типа Edition; метод get свойства возвращает объект типа Edition, данные которого совпадают с данными подобъекта базового класса, метод set присваивает значения полям из подобъекта базового класса.

В новой версии класса **Magazine** определить

* итератор с параметром типа double для перебора статей с рейтингом больше некоторого заданного значения;
* итератор с параметром типа string для перебора статей, в названии которых есть заданная строка.

В методе **Main()**

1. Создать два объекта типа Edition с совпадающими данными и проверить, что ссылки на объекты не равны, а объекты равны, вывести значения хэш-кодов для объектов.
2. В блоке try/catch присвоить свойству с тиражом издания некорректное значение, в обработчике исключения вывести сообщение, переданное через объект-исключение.
3. Создать объект типа Magazine, добавить элементы в списки статей и редакторов журнала и вывести данные объекта Magazine.
4. Вывести значение свойства типа Edition для объекта типа Magazine.
5. С помощью метода DeepCopy() создать полную копию объекта Magazine. Изменить данные в исходном объекте Magazine и вывести копию и исходный объект, полная копия исходного объекта должна остаться без изменений.
6. С помощью оператора foreach для итератора с параметром типа double вывести список всех статей с рейтингом больше некоторого заданного значения.
7. С помощью оператора foreach для итератора с параметром типа string вывести список статей, в названии которых есть заданная строка.

**Дополнительное задание: (5 баллов)**

В классе **Magazine**

* реализовать интерфейс System.Collections.IEnumerable для перебора статей (объектов типа Article), авторы которых не входят в список редакторов журнала; для этого определить вспомогательный класс MagazineEnumerator, реализующий интерфейс System.Collections.IEnumerator.
* определить итератор для перебора статей (объектов типа Article), авторы которых являются редакторами журнала, для этого определить метод, содержащий блок итератора и использующий оператор yield.
* определить итератор для перебора редакторов журнала (объектов типа Person), у которых нет статей в журнале, для этого определить метод, содержащий блок итератора и использующий оператор yield.

В методе **Main()**

1. С помощью оператора foreach для объекта типа Magazine вывести список статей, авторы которых не являются редакторами журнала.
2. С помощью оператора foreach для итератора, определенного в классе Magazine, вывести список статей, авторы которых являются редакторами журнала.
3. С помощью оператора foreach для итератора, определенного в классе Magazine, вывести список редакторов, у которых нет статей в журнале.

## Вариант 3. Требования к программе

Определить интерфейс

**interface INameAndCopy**

**{ string Name { get; set;}**

**object DeepCopy();**

**}**

Определить новые версии классов **Person**, **Paper** и **ResearchTeam**. Класс **ResearchTeam** определить как производный от класса **Team**. В классы **Team** и **ResearchTeam** добавить реализацию интерфейса INameAndCopy.

В классе **Paper** определить виртуальный метод object DeepCopy().

В новой версии класса **Person** дополнительно

* переопределить метод virtial bool Equals (object obj) и определить операции == и != так, чтобы равенство объектов типа Person трактовалось как совпадение всех данных объектов, а не ссылок на объекты Person;
* переопределить виртуальный метод int GetHashCode();
* определить виртуальный метод object DeepCopy().
* Определить класс **Team**. Класс **Team** имеет
* защищенное (protected) поле типа string с названием организации;
* защищенное поле типа int – регистрационный номер.

В классе **Team** определить:

* конструктор с параметрами типа string и int для инициализации полей класса;
* конструктор без параметров для инициализации по умолчанию;
* свойство типа string для доступа к полю с названием организации;
* свойство типа int для доступа к полю с номером регистрации; в методе set бросить исключение, если присваиваемое значение меньше или равно 0; при создании объекта-исключения использовать один из определенных в библиотеке CLR классов-исключений, инициализировать объект-исключение с помощью конструктора с параметром типа string.

В классе **Team**

* определить виртуальный метод object DeepCopy();
* реализовать интерфейс INameAndCopy.
* В классе **Team** переопределить (override):
* виртуальный метод virtial bool Equals (object obj) и определить операции == и != так, чтобы равенство объектов типа Team трактовалось как совпадение всех данных объектов, а не ссылок на объекты Team;
* виртуальный метод int GetHashCode();
* виртуальный метод string ToString() для формирования строки со значениями всех полей класса.

Новая версия класса **ResearchTeam** имеет базовый класс **Team** и следующие поля:

* закрытое поле типа string c названием темы исследований;
* закрытое поле типа TimeFrame с информацией о продолжительности исследований;
* закрытое поле типа System.Collections.ArrayList со списком участников проекта (объектов типа Person);
* закрытое поле типа System.Collections.ArrayList для списка публикаций (объектов типа Paper).

Код следующих конструкторов, методов и свойств из старой версии класса **ResearchTeam** необходимо изменить с учетом того, что часть полей класса перемещена в базовый класс **Team**, и в новой версии класса ResearchTeam для списка публикаций используется тип System.Collections.ArrayList:

* конструктор c параметрами типа string, string, int, TimeFrame для инициализации соответствующих полей класса;
* конструктор без параметров для инициализации по умолчанию;
* свойство типа System.Collections.ArrayList для доступа к полю со списком публикаций;
* свойство типа Paper (только с методом get), которое возвращает ссылку на публикацию с самой поздней датой выхода; если список публикаций пустой, свойство возвращает значение null;
* метод void AddPapers (params Paper[] ) для добавления элементов в список публикаций;
* перегруженная версия виртуального метода string ToString() для формирования строки со значениями всех полей класса, включая список публикаций и список участников проекта;
* метод string ToShortString(), который формирует строку со значениями всех полей класса без списка публикаций и списка участников проекта.

Дополнительно в новой версии класса **ResearchTeam** определить

* перегруженную версию виртуального метода object DeepCopy();
* свойство типа System.Collections.ArrayList для доступа к полю со списком участников проекта;
* метод void AddMembers ( params Person[] ) для добавления элементов в список участников проекта;
* свойство типа Team; метод get свойства возвращает объект типа Team, данные которого совпадают с данными подобъекта базового класса, метод set присваивает значения полям из подобъекта базового класса;
* реализовать интерфейс INameAndCopy.

В новой версии класса **ResearchTeam** определить

* итератор для последовательного перебора участников проекта (объектов типа Person), не имеющих публикаций;
* итератор с параметром типа int для перебора публикаций, вышедших за последние n лет, в котором число n передается через параметр итератора.

В методе **Main()**

1. Создать два объекта типа Team с совпадающими данными и проверить, что ссылки на объекты не равны, а объекты равны, вывести значения хэш-кодов для объектов.
2. В блоке try/catch присвоить свойству с номером регистрации некорректное значение, в обработчике исключения вывести сообщение, переданное через объект-исключение.
3. Создать объект типа ResearchTeam, добавить элементы в список публикаций и список участников проекта и вывести данные объекта ResearchTeam.
4. Вывести значение свойства Team для объекта типа ResearchTeam.
5. С помощью метода DeepCopy() создать полную копию объекта ResearchTeam. Изменить данные в исходном объекте ResearchTeam и вывести копию и исходный объект, полная копия исходного объекта должна остаться без изменений.
6. С помощью оператора foreach для итератора, определенного в классе ResearchTeam, вывести список участников проекта, которые не имеют публикаций.
7. С помощью оператора foreach для итератора с параметром, определенного в классе ResearchTeam, вывести список всех публикаций, вышедших за последние два года.

**Дополнительное задание: (5 баллов)**

В классе **ResearchTeam**

* реализовать интерфейс System.Collections.IEnumerable для перебора участников проекта (объектов типа Person), у которых есть публикации; для этого определить вспомогательный класс ResearchTeamEnumerator, реализующий интерфейс System.Collections.IEnumerator.
* определить итератор для перебора участников проекта (объектов типа Person), имеющих более одной публикации, для этого определить метод, содержащий блок итератора и использующий оператор yield.
* определить итератор для перебора публикаций (объектов типа Paper), вышедших за последний год, для этого определить метод, содержащий блок итератора и использующий оператор yield.

В методе **Main**()

1. С помощью оператора foreach для объекта типа ResearchTeam вывести список участников проекта, у которых есть публикации.
2. С помощью оператора foreach для итератора, определенного в классе ResearchTeam, вывести список участников проекта, имеющих более одной публикации.
3. C помощью оператора foreach для итератора, определенного в классе ResearchTeam, вывести список публикаций, вышедших за последний год.

# Практическая работа №6. Обобщения, коллекции

Цель: знакомство с универсальными типами, с классами-коллекций, оператор перебора коллекций, интерфейсы для работы с коллекциями. Понятия ковариантность и контрвариантность интерфейсов.

Когда вам необходимо создать коллекцию, то вы ее создаете с указанием типа данных, которые намерены в ней хранить. Если вам понадобится хранить данные другого типа, по вам приходиться вновь создавать коллекцию уже другого типа и соответственно переписывать методы длч работы с ней. Подобную проблему можно решить используя универсальные типы или обобщения.

Обобщения (или шаблоны) – это параметризированные типы.

Механизм параметризированных типов позволяет создавать классы, структуры, интерфейсы и методы, в которых используемые типы данных могут указываться в виде параметров.

Форма объявления параметризированного класса:

class Имя\_класса<список\_параметров\_типов>

{

// Здесь можно использовать типы из списка параметров

}

Когда у имени класса после имени стоят треугольные скобки с параметром (T или др.), такой класс называется обобщённым классом. По аналогии с классом, бывают обобщённые структуры, обобщённые интерфейсы и обобщённые методы.

В случае с обобщённым методом тип-параметр также указываем в треугольных скобках после имени метода:

public int GenericMethod<T>(float param1, bool param2) {}

Когда тип описан параметризированным, внутри него можно использовать этот тип T (или другое имя), как будто это обычный определённый тип.

// Обобщённый класс MyGeneric

class MyGeneric<T>

{

// Свойство целочисленного типа

public int IntProperty { get; set; }

// Поле типа-параметра

private T field;

// Свойство типа-параметра для этого поля

public T Property

{

get { return field; }

set { field = value; }

}

// Конструктор, принимающий число и объект типа-параметра

public MyGeneric(int a, T b)

{

IntProperty = a;

Property = b;

}

// Метод, возвращающий объект типа-параметра

public T PowerfulMethod(char a)

{

return Property;

}

List<int> ls = new List<int>();

Dictionary<string, string> dict = new Dictionary<string, string>();

Stack<double> st1 = new Stack<double>();

Обобщённые методы могут быть определены внутри необобщённого класса.

Сделав интерфейс обобщённым, можно, например, описать в нём такой метод, который будет в качестве типа возвращаемого значения иметь тип, в котором этот интерфейс реализовывается.

// Обобщённый интерфейс с типом-параметром T

// С ограничением, что тип T должен быть классом, реализующим этот интерфейс

interface ICloneable<T> where T : class, ICloneable<T>

{

// Метод возвращает объект типа T

public T Clone();

}

// Класс Foo, который реализует этот обобщённый интерфейса и в качестве типа-параметра использует свой тип Foo

class Foo : ICloneable<Foo>

{

// Значит, этот метод должен возвращать объект этого же типа Foo

public Foo Clone()

{ }

}

**Ковариантность и контравариантность**

Понятие ковариантности связано с возможностью методов обобщённого интерфейса использовать значения разных типов, связанных между собой наследованием.

Есть 3 варианта этого поведения:

* инвариантность (по умолчанию);
* ковариантность (ключевое слово out);
* контравариантность (ключевое слово in).

Упрощая, можно сказать, что ковариантность позволяет настраивать полиморфизм для обобщённых интерфейсов (и делегатов).

**Инвариантность** обобщённых интерфейсов заключается в отсутствии возможности использовать полиморфизм для конкретных типов со связанными типами параметрами.

// Класс Банковского счёта

class Account

{

public decimal Balance { get; set; } // Свойсто Баланс на счету

public void DoTransfer(decimal sum) // Метод Перевода суммы на этот счёт

{

Console.WriteLine("Transfering {0}$....", sum);

Balance += sum;

}

}

// Производный класс от Банковского счёта - депозит

class DepositAccount : Account

{

// ...

}

// Обобщённый интерфейс Банка с типом-параметром T

interface IBank<T>

{

T CreateAccount(int sum); // Метод для создания аккаунта, возвращает объект типа T

}

// Обобщённый класс Банк с типом-параметром T

// Реализует интерфейс Банка с этим же типом-параметром

// И на тип-параметр наложены ограничение на наличие конструктора по умолчанию

// и соответсвие классу Account, либо производным от него классам

class Bank<T> : IBank<T> where T : Account, new()

{

// Реализация метода CreateAccount из интерфейса IBank

public T CreateAccount(int sum)

{

T acc = new T();// Создаём новый счёт

acc.DoTransfer(sum); // Выполняем начальный перевод

return acc; // Возвращаем объект счёта

}

}

static void Main(string[] args)

{

// Создадим объект Банка для типа DepositAccount

// Полученный класс будет реализовывать интерфейс IBank<DepositAccount>

IBank<DepositAccount> depositBank = new Bank<DepositAccount>();

Account acc1 = depositBank.CreateAccount(34);

// Ошибка

// Мы не можем представить этот объект в виде интерфейса IBank<Account>

// Потому что depositBank реализует интерфейс IBank<DepositAccount>

// А IBank<Account> и IBank<DepositAccout> - это 2 разных интерфейса

IBank<Account> ordinaryBank = depositBank;

Account acc2 = ordinaryBank.CreateAccount(45);

}

}

**Ковариантность** интерфейса позволяет представлять объекты с интерфейсом с более конкретным типом в виде интерфейсов с менее конкретным типом. Ковариантные типы в методах интерфейса можно использовать только для возвращаемых значений в методах.

class Account

{

public decimal Balance { get; set; }

public void DoTransfer(decimal sum)

{

Console.WriteLine("Transfering {0}$....", sum);

Balance += sum;

}

}

class DepositAccount : Account

{ …}

// Интерфейс с ковариантным параметром типа

interface IBank<out T>

{

T CreateAccount(int sum);

}

class Bank<T> : IBank<T> where T : Account, new()

{

public T CreateAccount(int sum)

{

T acc = new T();

acc.DoTransfer(sum);

return acc;

}

}

static void Main(string[] args)

{

IBank<DepositAccount> depositBank = new Bank<DepositAccount>();

Account acc1 = depositBank.CreateAccount(34);

// Теперь всё работает - это эффект ковариантности

IBank<Account> ordinaryBank = depositBank;

Account acc2 = ordinaryBank.CreateAccount(45);

}

**Контравариантность** интерфейса, наоборот, позволяет представлять объекты с интерфейсом с менее конкретным типом в виде интерфейсов с более конкретным типом. Допускает использование контравариантных типов только в параметрах методов интерфейса.

// Обобщённый интерфейс с контравариантным параметром типа

interface ITransaction<in T>

{

// Метод принимает объект контравариантного типа

void DoOperation(T account, int sum);

}

// Обобщённый класс, реализующий интерфейс ITransaction для тип T

// С ограничением, что тип T должен быть потомком класса Account

class Transaction<T> : ITransaction<T> where T : Account

{

// Реализация метода из интерфейса

public void DoOperation(T account, int sum)

{

account.DoTransfer(sum);

}

}

static void Main(string[] args)

{

// Создаём класс Transaction для типа Account

ITransaction<Account> accTransaction = new Transaction<Account>();

// И передаём в него объект класса Account

accTransaction.DoOperation(new Account(), 400);

// Также создаём класс Transaction для типа Account,

// Но записываем его в тип интерфейса ITransaction для более конкретного типа DepositAccount

ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();

// Хоть объект и был создан для типа Account

// За счёт контравариантности мы можем передавать в его метод объекты типа DepositAccount

depAccTransaction.DoOperation(new DepositAccount(), 450);

}

## Задание к практической работе №6. (8 баллов)

Во всех вариантах работы требуется определить класс **TestCollections**, который содержит поля следующих типов

* System.Collections.Generic.List<TKey> ;
* System.Collections.Generic.List<string> ;
* System.Collections.Generic.Dictionary<TKey, TValue> ;
* System.Collections.Generic.Dictionary<string, TValue> .

Конкретные значения типовых параметров TKey и TValue зависят от варианта. Во всех вариантах тип ключа TKey и тип значения TValue связаны отношением базовый-производный. Во всех вариантах в классе TValue определено свойство, которое возвращает ссылку на объект типа TKey с данными, совпадающими с данными подобъекта базового класса (это свойство должно возвращать ссылку на объект типа TKey, а не ссылку на вызывающий объект TValue).

В конструкторе класса **TestCollections** создаются коллекции с заданным числом элементов. Надо сравнить время поиска элемента в коллекциях-списках List<TKey> и время поиска элемента по ключу и элемента по значению в коллекциях-словарях Dictionary<TKey,TValue>.

Для автоматической генерации элементов коллекций в классе **TestCollections** надо определить статический метод, который принимает один целочисленный параметр типа int и возвращает ссылку на объект типа TValue.

Каждый объект TValue содержит подобъект базового класса TKey. Соответствие между значениями целочисленного параметра метода и подобъектами TKey класса TValue должно быть взаимно-однозначным – равным значениям параметра должны отвечать равные объекты TKey и наоборот. Равенство объектов типа TKey трактуется так же, как это было сделано в лабораторной работе 5 при определении операций равенства объектов.

Все четыре коллекции содержат одинаковое число элементов. Каждому элементу из коллекции List<TKey> должен отвечать элемент в коллекции Dictionary<TKey, TValue> с равным значением ключа. Список List<string> состоит из строк, которые получены в результате вызова метода ToString() для объектов TKey из списка List<TKey>. Каждому элементу списка List<string> отвечает элемент в коллекции-словаре Dictionary<string, TValue> с равным значением ключа типа string.

Число элементов в коллекциях вводится пользователем в процессе работы приложения. Если при вводе была допущена ошибка, приложение должно обработать исключение, сообщить об ошибке ввода и повторить прием ввода до тех пор, пока не будет правильно введено целочисленное значение.

Для четырех разных элементов – первого, центрального, последнего и элемента, не входящего в коллекцию – надо измерить время поиска

* элемента в коллекциях List<TKey> и List<string> с помощью метода Contains;
* элемента по ключу в коллекциях Dictionary< TKey, TValue> и Dictionary <string, TValue > с помощью метода ContainsKey;
* значения элемента в коллекции Dictionary< TKey, TValue > с помощью метода ContainsValue.

Так как статический метод для автоматической генерации элементов должен обеспечивать взаимно-однозначное соответствие между значением целочисленного параметра метода и объектами TKey, этот метод можно использовать как при создании коллекций с большим числом элементов, так и для генерации элемента для поиска.

## Вариант 1.

Определить новые версии классов **Person** и **Student**

В класс **Person** добавить реализацию интерфейсов

* System.IComparable для сравнения объектов типа Person по полю с фамилией;
* System.Collections.Generic.IComparer<Person> для сравнения объектов типа Person по дате рождения.

В новой версии класса **Student** для списков зачетов и экзаменов использовать типы

* System.Collections.Generic.List<Test> для списка зачетов;
* System.Collections.Generic.List<Exam> для списка экзаменов.

В новой версии класса **Student** сохранить все остальные поля, свойства и методы из предыдущей версии класса, внести необходимые исправления в код свойств и методов из-за изменения типов полей для списков зачетов и экзаменов.

Определить **вспомогательный класс,** реализующий интерфейс System.Collections.Generic.IComparer<Student>, который можно использовать для сравнения объектов типа Student по среднему баллу.

Определить класс **StudentCollection**, который содержит

* закрытое поле типа System.Collections.Generic.List<Student>;
* метод void AddDefaults(), c помощью которого можно добавить некоторое число элементов типа Student для инициализации коллекции по умолчанию;
* метод void AddStudents (params Student[] ) для добавления элементов в список List<Student>;
* перегруженную версию виртуального метода string ToString() для формирования строки c информацией обо всех элементах списка List<Student>, включающую значения всех полей, список зачетов и экзаменов для каждого элемента Student;
* метод string ToShortString(), который формирует строку c информацией обо всех элементах списка List<Student>, содержащую значения всех полей, средний балл, число зачетов и число экзаменов для каждого элемента Student, но без списков зачетов и экзаменов.

В классе **StudentCollection** определить методы, выполняющие сортировку списка List<Student>

* по фамилии студента с использованием интерфейса IComparable, реализованного в классе Person;
* по дате рождения студента с использованием интерфейса IComparer<Person>, реализованного в классе Person;
* по среднему баллу с использованием интерфейса IComparer<Student>, реализованного во вспомогательном классе.

В классе **StudentCollection** определить свойства и методы, выполняющие операции со списком List<Student>:

* свойство типа double (только с методом get), возвращающее максимальное значение среднего балла для элементов списка List<Student>; если в коллекции нет элементов, свойство возвращает некоторое значение по умолчанию;
* свойство типа IEnumerable<Student> (только с методом get), возвращающее подмножество элементов списка List<Student> с формой обучения Education.Specialist;
* метод List<Student> AverageMarkGroup(double value), который возвращает список, в который входят элементы Student из списка List<Student> с заданным значением среднего балла;

Определить класс **TestCollections**, в котором в качестве типа TKey используется класс Person, а в качестве типа TValue - класс Student. Класс содержит закрытые поля с коллекциями типов

* System.Collections.Generic.List<Person>;
* System.Collections.Generic.List<string>;
* System.Collections.Generic.Dictionary <Person, Student>;
* System.Collections.Generic.Dictionary <string, Student>.

В классе **TestCollections** определить

* статический метод с одним целочисленным параметром типа int, который возвращает ссылку на объект типа Student и используется для автоматической генерации элементов коллекций;
* конструктор c параметром типа int (число элементов в коллекциях) для автоматического создания коллекций с заданным числом элементов;
* метод, который вычисляет время поиска элемента в списках List<Person> и List<string>, время поиска элемента по ключу и время поиска элемента по значению в коллекциях-словарях Dictionary<Person, Student> и Dictionary<string, Student>.

В методе **Main()**

1. Создать объект типа StudentCollection. Добавить в коллекцию несколько различных элементов типа Student и вывести объект StudentCollection.

2. Для созданного объекта StudentCollection вызвать методы, выполняющие сортировку списка List<Student> по разным критериям, и после каждой сортировки вывести данные объекта. Выполнить сортировку

* по фамилии студента;
* по дате рождения;
* по среднему баллу.

3. Вызвать методы класса StudentCollection, выполняющие операции со списком List<Student>, и после каждой операции вывести результат операции. Выполнить

* вычисление максимального значения среднего балла для элементов списка;
* фильтрацию списка для отбора студентов с формой обучения Education.Specialist;
* группировку элементов списка по значению среднего балла; вывести все группы элементов.

4. Создать объект типа TestCollections. Вызвать метод для поиска в коллекциях первого, центрального, последнего и элемента, не входящего в коллекции. Вывести значения времени поиска для всех четырех случаев. Вывод должен содержать информацию о том, к какой коллекции и к какому элементу относится данное значение.

## Вариант 2.

Определить новые версии классов **Edition** и **Magazine.** В новой версии класса **Magazine** использовать типы

* System.Collections.Generic.List<Person> для списка редакторов журнала;
* System.Collections.Generic.List<Article> для списка статей в журнале.

В новых версиях классов **Edition** и **Magazine** сохранить все остальные поля, свойства и методы из предыдущей версии класса, внести необходимые исправления в код свойств и методов из-за изменения типов полей для списка редакторов и списка статей.

В класс **Edition** добавить реализацию

* интерфейсa System.IComparable для сравнения объектов Edition по полю с названием издания;
* интерфейсa System.Collections.Generic.IComparer<Edition> для сравнения объектов Edition по дате выхода издания.

Определить **вспомогательный класс**, реализующий интерфейс System.Collections.Generic.IComparer<Edition>, который можно использовать для сравнения объектов типа Edition по тиражу издания.

Определить класс **MagazineCollection**, который содержит

* закрытое поле типа System.Collections.Generic.List<Magazine>;
* метод void AddDefaults (), c помощью которого в список List<Magazine> можно добавить некоторое число элементов типа Magazine для инициализации коллекции по умолчанию;
* метод void AddMagazines (params Magazine []) для добавления элементов в список List<Magazine>;
* перегруженную версию виртуального метода string ToString() для формирования строки с информацией обо всех элементах списка List<Magazine>, в том числе значения всех полей, список редакторов журнала и список статей в журнале для каждого элемента Magazine;
* виртуальный метод string ToShortString(), который формирует строку с информацией обо всех элементах списка List<Magazine>, содержащую значения всех полей, средний рейтинг статей, число редакторов журнала и число статей в журнале для каждого элемента Magazine, но без списков редакторов и статей.

В классе **MagazineCollection** определить свойства и методы, выполняющие сортировку списка List<Magazine>

* по названию издания с использованием интерфейса IComparable, реализованного в классе Edition;
* по дате выхода издания с использованием интерфейса IComparer<Edition>, реализованного в классе Edition;
* по тиражу издания с использованием интерфейса IComparer<Edition>, реализованного во вспомогательном классе.

В классе **MagazineCollection** определить методы, выполняющие операции со списком List<Magazine>:

* свойство типа double (только с методом get), возвращающее максимальное значение среднего рейтинга статей для элементов списка List<Magazine>; если в коллекции нет элементов, свойство возвращает некоторое значение по умолчанию;
* свойство типа IEnumerable<Magazine> (только с методом get), возвращающее подмножество элементов списка List<Magazine> с периодичностью выхода журнала Frequency.Monthly;
* метод List<Magazine> RatingGroup(double value), который возвращает список, содержащий элементы Magazine из List<Magazine> со средним рейтингом статей, который больше или равен value.

Определить класс **TestCollections**, в котором в качестве типа TKey используется класс Edition, а в качестве типа TValue - класс Magazine. Класс содержит закрытые поля с коллекциями типов

* System.Collections.Generic.List<Edition>;
* System.Collections.Generic.List<string>;
* System.Collections.Generic.Dictionary <Edition, Magazine>;
* System.Collections.Generic.Dictionary <string, Magazine>.

В классе **TestCollection** определить

* статический метод с одним целочисленным параметром типа int, который возвращает ссылку на объект типа Magazine и используется для автоматической генерации элементов коллекций;
* конструктор c параметром типа int (число элементов в коллекциях) для автоматического создания коллекций с заданным числом элементов;
* метод, который вычисляет время поиска элемента в списках List<Edition> и List<string>, время поиска элемента по ключу и время поиска элемента по значению в коллекциях-словарях Dictionary< Edition, Magazine> и Dictionary<string, Magazine>.

В методе **Main()**

1. Создать объект типа MagazineCollection. Добавить в коллекцию несколько элементов типа Magazine с разными значениями полей и вывести объект MagazineCollection.

2. Для созданного объекта MagazineCollection вызвать методы, выполняющие сортировку списка List<Magazine> по разным критериям, и после каждой сортировки вывести данные объекта. Выполнить сортировку

* по названию издания;
* по дате выхода издания;
* по тиражу издания.

3. Вызвать методы класса MagazineCollection, выполняющие операции со списком List<Magazine>, и после каждой операции вывести результат операции. Выполнить

* вычисление максимального значения среднего рейтинга статей для элементов списка; вывести максимальное значение;
* фильтрацию списка для отбора журналов с периодичностью выхода Frequency.Monthly, вывести результат фильтрации;
* группировку элементов списка по значению среднего рейтинга статей; вывести все группы элементов.

4. Создать объект типа TestCollections. Вызвать метод для поиска в коллекциях первого, центрального, последнего и элемента, не входящего в коллекции. Вывести значения времени поиска для всех четырех случаев. Вывод должен содержать информацию о том, к какой коллекции и к какому элементу относится данное значение.

## Вариант 3.

Определить новые версии классов **Team** и **ResearchTeam**

В новой версии класса **ResearchTeam** использовать типы

* System.Collections.Generic.List<Person>для списка участников проекта;
* System.Collections.Generic.List<Paper> для списка публикаций;

В новых версиях классов **Team** и **ResearchTeam** сохранить все остальные поля, свойства и методы из предыдущих версий, внести необходимые исправления в код свойств и методов из-за изменения типа полей для списков.

В новую версию класса **Team** добавить реализацию интерфейсa System.IComparable для сравнения объектов Team по полю с номером регистрации.

В новую версию класса **ResearchTeam** добавить реализацию интерфейсa System.Collections.Generic.IComparer<ResearchTeam> для сравнения объектов ResearchTeam по названию темы исследований.

Определить **вспомогательный класс**, реализующий интерфейс System.Collections.Generic.IComparer<ResearchTeam>, который можно использовать для сравнения объектов типа ResearchTeam по числу публикаций.

Определить класс **ResearchTeamCollection**, который содержит

* закрытое поле типа System.Collections.Generic.List<ResearchTeam>;
* метод void AddDefaults (), c помощью которого в список List<ResearchTeam> можно добавить некоторое число элементов типа ResearchTeam для инициализации коллекции по умолчанию;
* метод void AddResearchTeams ( params ResearchTeam []) для добавления элементов в список List<ResearchTeam>;
* перегруженную версию виртуального метода string ToString() для формирования строки c информацией обо всех элементах списка List<ResearchTeam>, которая содержит значения всех полей, список участников проекта и список публикаций для каждого элемента ResearchTeam;
* виртуальный метод string ToShortString(), который формирует строку с информацией обо всех элементах списка List<ResearchTeam>, включающую значения всех полей, число участников проекта и число публикаций для каждого элемента ResearchTeam, но без списков участников и публикаций.

В классе **ResearchTeamCollection** определить методы, выполняющие сортировку списка List<ResearchTeam>

* по номеру регистрации с использованием интерфейса IComparable, реализованного в классе Team;
* по названию темы исследований с использованием интерфейса IComparer<ResearchTeam>, реализованного в классе ResearchTeam;
* по числу публикаций с использованием интерфейса IComparer<ResearchTeam>, реализованного во вспомогательном классе.

В классе **ResearchTeamCollection** определить свойства и методы, выполняющие операции со списком List<ResearchTeam>:

* свойство типа int (только с методом get), возвращающее минимальное значение номера регистрации для элементов списка List<ResearchTeam>; если в коллекции нет элементов, свойство возвращает некоторое значение по умолчанию;
* свойство типа IEnumerable<ResearchTeam> (только с методом get), возвращающее подмножество элементов списка List<ResearchTeam> с продолжительностью исследований TimeFrame.TwoYears;
* метод List<ResearchTeam> NGroup(int value), который возвращает список, в который входят элементы ResearchTeam из списка List<ResearchTeam> с заданным числом участников исследования.

Определить класс **TestCollections**, в котором в качестве типа TKey используется класс Team, а в качестве типа TValue - класс ResearchTeam. Класс содержит закрытые поля с коллекциями типов

* System.Collections.Generic.List<Team>;
* System.Collections.Generic.List<string>;
* System.Collections.Generic.Dictionary <Team, ResearchTeam>;
* System.Collections.Generic.Dictionary <string, ResearchTeam>.

В классе **TestCollection**s определить

* статический метод с одним целочисленным параметром типа int, который возвращает ссылку на объект типа ResearchTeam и используется для автоматической генерации элементов коллекций;
* конструктор c параметром типа int (число элементов в коллекциях) для автоматического создания коллекций с заданным числом элементов;
* метод, который вычисляет время поиска элемента в списках List<Team> и List<string>, время поиска элемента по ключу и время поиска значения элемента в коллекциях-словарях Dictionary<Team, ResearchTeam> и Dictionary <string, ResearchTeam>.

В методе **Main()**

1. Создать объект типа ResearchTeamCollection. Добавить в коллекцию несколько элементов типа ResearchTeam с разными значениями полей и вывести объект ResearchTeamCollection.

2. Для созданного объекта ResearchTeamCollection вызвать методы, выполняющие сортировку списка List<ResearchTeam> по разным критериям, и после каждой сортировки вывести данные объекта. Выполнить сортировку

* по номеру регистрации;
* по названию темы исследований;
* по числу публикаций.

3. Вызвать методы класса ResearchTeamCollection, выполняющие операции со списком List<ResearchTeam>, и после каждой операции вывести результат операции. Выполнить

* вычисление минимального значения номера регистрации для элементов списка; вывести минимальное значение;
* фильтрацию проектов с продолжительностью исследований TimeFrame.TwoYears, вывести результат фильтрации;
* группировку элементов списка по числу публикаций; вывести все группы элементов из списка.

4. Создать объект типа TestCollections. Вызвать метод для поиска в коллекциях первого, центрального, последнего и элемента, не входящего в коллекции. Вывести значения времени поиска для всех четырех случаев. Вывод должен содержать информацию о том, к какой коллекции и к какому элементу относится данное значение.

# Практическая работа №7. Делегаты. События.

Цель: создание и работа с делегатами, событиями, лямбда–выражения и лямбда-операторы. Вариативность делегатов, обобщенные делегаты.

**Делегатом** называют особый тип данных, который может ссылаться на метод. Формат создания делегата подобен описанию интерфейса. Только речь идет не о возможности наследования в разных классах сигнатуры метода без описания его реализации, а о создании объекта делегата, содержащего ссылку на конкретный метод с описанной реализацией. Под сигнатурой будем понимать шаблон метода, включающий тип возвращаемого функцией значения и типы данных аргументов, например: – double (int, byte), – void (string), – bool (int). Таким образом, объект делегата содержит не только адрес метода, но и аргументы (при необходимости) и возвращаемое значение. Самое важное заключается в том, что этот метод можно вызывать по ссылке, то есть именно делегат позволяет вызывать метод, на который он ссылается. Еще одно замечание о способностях делегатов состоит в их способности менять ссылку на конкретный метод во время выполнения программы, то есть динамически, а не только на этапе разработки.

Подробное рассмотрение делегатов см.в лекции «Делегаты».

Делегат создаётся, как и любой объект, через описание типа и имени. При создании у делегата нет значения (null). А значениями для делегата служат методы. Делегату может быть назначен не один метод, а любое количество. Это позволяет создаватьтак называемую цепочку вызовов: при вызове делегата, методы, назначенные ему, последовательно вызываются друг за другом.

Для добавления дополнительных методов к делегату используются операторы + и +=.

А для удаления делегата применяются операторы - и -=.

**Ковариантность и контравариантность делегатов**

У делегатов тоже есть эти страшные свойства. Только, в отличие от обобщённых интерфейсов, для делегатов они действуют всегда по умолчанию.

Свойство **ковариантности** позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.

Свойство **контравариантности** позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

// Класс Здание

class Building

{ ... }

class School : Building // Производный от него класс Школа

{ ... }

class Hotel : Building // Производный от здания класс Отель

{ ... }

class ConstructCompany // Класс строительной компании

{

// Метод Постройки школы, возвращающий новый объект школы

public School BuildSchool()// Создаётся и возвращается объект класса School

{

Console.WriteLine("Construction company is building a school");

return new School();

}

// Метод Постройки отеля, возвращающий объект отеля

public Hotel BuildHotel()// Создаётся и возвращается объект Hotel

{

Console.WriteLine("Construction company is building a new hotel");

return new Hotel();

}

}

class Citizen // Класс Горожанин

{

// Метод Посещения постройки, принимает постройку в качестве параметра

public void VisitBuilding(Building someBuilding)

{

Console.WriteLine("The citizen decided to visit the Building");

}

// Методы Посещения отеля, принимает объект класса Hotel

public void VisitHotel(Hotel hotel)

{

Console.WriteLine("The citizen decided to visit the Hotel");

}

}

// Делегат для демонстрации ковариантности. Возвращаемым типом указан Building

delegate Building CreateBuilding();

// Делегат для демонстрации контравариантности. Принимает аргумент типа Hotel

delegate void Visit(Hotel building);

static void Main(string[] args)

{

ConstructCompany company = new ConstructCompany();// Создаём строительную компанию

CreateBuilding build; // Создаём делегат типа CreateBuilding

Random randGen = new Random();

int randNum = randGen.Next(2);

// 50 на 50

if (randNum == 0)

{

// Что назначит делегату метод BuildSchool

// Хотя у делегата указано, что тип возвращаемого значения должен быть Building

// Мы можем назначить метод, который возвращает School - производный от указанного класса Building. Это и есть ковариантность

build = company.BuildSchool;

}

else

{

// Или метод BuildHotel

build = company.BuildHotel;

}

// Вызываем делегат и сохраняем полученную постройку в переменную b

Building b = build();

Citizen adler = new Citizen();// Создаём горожанина

// Создаём делегат типа Visit и назначаем ему метод VisitBuilding

// Хотя у делегата отмечено, что он должен принимать параметр типа Hotel

// Мы можем назначит ему метод, принимающий параметр типа Building - базовый для класса Hotel

// Это - контравариантность

Visit visit = adler.VisitBuilding;

// А потом ещё раз (можно несколько раз назначать делегату один метод)

// Тогда он просто вызовется несколько раз

visit += adler.VisitBuilding;

visit += adler.VisitHotel; // Также добавляем к делегату метод VisitHotel,

visit += adler.VisitBuilding; // И ещё один VisitBuilding

// Если строительная компания построила отель, а не школу

if (b is Hotel hot)

{

// Вызываем делегат, и передаём в качестве параметра объект отеля hot

// Полученный путём преобразования оператора is в условии проверки if

visit(hot);

}

}

Делегаты можно использовать в качестве аргументов методов, а также в качестве возвращаемых значений.

И это очень распространённый, гибкий и мощный инструмент.

В аргумент метода типа делегата может быть передан как непосредственно делегат такого типа, так и просто метод подходящего описания.

**Лямбда-выражения**

Для описания Лямбда-выражения используется лямбда-оператор =>.

Бывает 2 вида лямбда-выражений:

* одиночные;
* блочные.

Одиночное лямбда выражение используется, когда действие может быть описано одним выражением.

параметр => выражение

или

(список\_параметров) => выражение

Одиночные лямбды по умолчанию возвращают результат описанного в них выражения, поэтому если в анонимной функции предполагался return – его писать не нужно.

Внутри блочных лямбда-выражений может содержаться любое количество выражений, любой код, т.е. нет никаких ограничений. На за счёт этого приходится расплачиваться красотой синтаксиса:

параметр => { /\* ... \*/ }

(список\_параметров) => { /\* ... \*/ return /\* ... \*/ }

Встроенные делегаты

Вообще, в .NET есть набор встроенных обобщѐнных типов делегатов, которые можно применять практически в любой ситуации, когда нужны делегаты. Т.е. нет необходимости объявлять свои типы делегатов.

Встроенные типы делегатов:

* Action;
* Predicate;
* Func.

**Делегаты Action**

Встроенный делегат Action принимает до 16 параметров любых типов и не имеет возвращаемого значения.

public delegate void Action();

public delegate void Action<in T1>(T1 obj1);

public delegate void Action<in T1, in T2>(T1 obj1, T2 obj2); // И т.д.

**пример:**

static void Operation(int x1, int x2, Action<int, int> op)

{

if (x1 > x2)

{

// Вызываем делегат из третьего параметра, передавая ему эти 2 числа x1 и x2

op(x1, x2);

}

}

static void Main(string[] args)

{

// Создаём объект делегата Action<int, int>

Action<int, int> op;

// Назначаем ему лямбда-выражение, выводящее на консоль сумму параметров

op = (x, y) => Console.WriteLine("Сумма чисел: " + (x + y));

// Вызываем статический метод Operation, передавая 2 числа и делегат op

Operation(10, 6, op);

// Переназначаем делегату новое лямбда-выражение для разности чисел

op = (x, y) => Console.WriteLine("Разность чисел: " + (x1 - x2));

// Вызываем метод Operation, передаём делегат op

Operation(10, 6, op);

}

**Делегат Predicate**

Обобщённый делегат Predicate существует лишь в одной вариации, и принимает параметр любого типа, а возвращает bool.

Как правило, он используется для различных методов фильтрации коллекций, например, поиска элементов по заданному предикату.

public delegate bool Predicate<in T>(T obj);

Теперь перейдем к ещё одному важному способу использования делегатов, к ***событиям***, на которых и основано всё событийное программирование. С помощью делегатов можно создавать механизм обратных вызовов в программе. Однако C# для той же цели предоставляет более удобные и простые конструкции под названием события, которые сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова **event**, после которого идет название делегата. Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры и возвращать тот же тип, что и делегат.

Итак, посмотрим на примере. Для этого создадим класс **Account**

class Account

{

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrowed;

// Событие, возникающее при добавление на счет

public event AccountStateHandler Added;

int \_sum; // Переменная для хранения суммы

int \_percentage; // Переменная для хранения процента

public Account(int sum, int percentage)

{

\_sum = sum;

\_percentage = percentage;

}

public int CurrentSum

{

get { return \_sum; }

}

public void Put(int sum)

{

\_sum += sum;

if (Added != null)

Added("На счет поступило " + sum);

}

public void Withdraw(int sum)

{

if (sum <= \_sum)

{ \_sum -= sum;

if (Withdrowed != null)

Withdrowed("Сумма " + sum + " снята со счета");

}

else

{

if (Withdrowed != null)

Withdrowed("Недостаточно денег на счете");

}

}

public int Percentage

{

get { return \_percentage; }

}

}

Здесь определены два события – **Withdrowed** и **Added**, которые объявлены как экземпляры делегата **AccountStateHandler**. Поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах **Put** и **Withdraw** вызываем эти события. Перед вызовом проверяем, закреплены ли за ними обработчики:

if (Withdrowed != null)

Так как эти события представляют делегат **AccountStateHandler**, принимающий в качестве параметра строку, то и при вызове событий передаем в них строку. Теперь используем события в основной программе:

static void Main(string[] args)

{

Account account = new Account(200, 6);

// Добавляем обработчики события

account.Added += Show\_Message;

account.Withdrowed += Show\_Message;

account.Withdraw(100);

// Удаляем обработчик события

account.Withdrowed -= Show\_Message;

account.Withdraw(50);

account.Put(150);

Console.ReadLine();

}

private static void Show\_Message(string message)

{

Console.WriteLine(message);

}

В итоге получим следующий консольный вывод:

Сумма 100 снята со счета

На счет поступило 150

## Задание на практическую работу №7 (8баллов)

**Информация для всех вариантов**

В работе требуется определить класс, содержащий типизированную коллекцию, который с помощью событий извещает об изменениях в коллекции.

Коллекция состоит из объектов ссылочных типов. Коллекция изменяется при удалении/добавлении элементов или при изменении одной из входящих в коллекцию ссылок, например, когда одной из ссылок присваивается новое значение. В этом случае в соответствующих методах или свойствах класса бросаются события.

При изменении данных объектов, ссылки на которые входят в коллекцию, значения самих ссылок не изменяются. Этот тип изменений не порождает событий.

Для событий, извещающих об изменениях в коллекции, определяется свой делегат. События регистрируются в специальных классах-слушателях.

## Вариант 1. Требования к программе

Определить новую версию класса **StudentCollection**, которая с помощью событий сообщает об изменениях в коллекции.

Для событий определить делегат **StudentListHandler** с сигнатурой:

**void StudentListHandler**

**(object source, StudentListHandlerEventArgs args);**

Класс **StudentListHandlerEventArgs**, производный от класса System.EventArgs, содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство типа Student для ссылки на объект, с которым связаны изменения;
* конструкторы для инициализации класса;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех полях класса.

В новую версию класса **StudentCollection** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Remove (int j) для удаления элемента с номером j из списка List<Student>; если в списке нет элемента с номером j, метод возвращает значение false;
* индексатор типа Student (с методами get и set) с целочисленным индексом для доступа к элементу списка List<Student> с заданным номером.

В новую версию класса **StudentCollection** добавить два события типа StudentListHandler

* **StudentsCountChanged**, которое происходит при добавлении нового элемента в коллекцию или при удалении элемента из коллекции; через объект StudentListHandlerEventArgs cобытие передает имя коллекции, строку с информацией о том, что в коллекцию был добавлен новый элемент или из нее был удален элемент, ссылку на добавленный или удаленный элемент Student;
* **StudentReferenceChanged**, которое происходит, когда одной из ссылок, входящих в коллекцию, присваивается новое значение; через объект StudentListHandlerEventArgs событие передает имя коллекции, строку с информацией о том, что был заменен элемент в коллекции, и ссылку на новый элемент Student.

Событие StudentsCountChanged бросают следующие методы класса StudentCollection

* AddDefaults();
* AddStudents (params Student[] ) ;
* Remove (int j).

Событие StudentReferenceChanged бросает метод set индексатора, определенного в классе StudentCollection.

Определить класс **Journal**, который можно использовать для накопления информации об изменениях в коллекциях типа StudentCollection. Класс Journal хранит информацию в списке объектов типа **JournalEntry**. Каждый элемент списка содержит информацию об отдельном изменении, которое произошло в коллекции.

Класс **JournalEntry** содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство типа string c данными объекта Student, с которым связаны изменения в коллекции;
* конструктор для инициализации полей класса;
* перегруженную версию метода string ToString().

Класс **Journal** содержит

* закрытое поле типа System.Collections.Generic.List<JournalEntry>;
* обработчики событий StudentsCountChanged и StudentReferenceChanged, которые добавляют новый элемент JournalEntry в список List<JournalEntry>; для инициализации JournalEntry используется информация из объекта StudentListHandlerEventArgs, который передается вместе с событием;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<JournalEntry>.

В методе **Main()**

1. Создать две коллекции StudentCollection.
2. Создать два объекта типа Journal, один объект Journal подписать на события StudentsCountChanged и StudentReferenceChanged из первой коллекции StudentCollection, другой объект Journal подписать на события StudentReferenceChanged из обеих коллекций StudentCollection.
3. Внести изменения в коллекции StudentCollection

* добавить элементы в коллекции;
* удалить некоторые элементы из коллекций;
* присвоить некоторым элементам коллекций новые значения.

1. Вывести данные обоих объектов Journal.

## Вариант 2. Требования к программе

Определить новую версию класса **MagazineCollection**, которая с помощью событий сообщает об изменениях в коллекции.

Для событий определить делегат **MagazineListHandler** с сигнатурой:

**void MagazineListHandler**

**(object source, MagazineListHandlerEventArgs args);**

Класс **MagazineListHandlerEventArgs**, производный от класса System.EventArgs, содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство типа int с номером элемента, который был изменен;
* конструкторы для инициализации класса;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех полях класса.

В новую версию класса **MagazineCollection** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Replace (int j, Magazine mg) для замены элемента с номером j из списка List<Magazine> на элемент mg; если в списке нет элемента с номером j, метод возвращает значение false;
* индексатор типа Magazine (с методами get и set) с целочисленным индексом для доступа к элементу списка List<Magazine> с заданным номером.

В новую версию класса **MagazineCollection** добавить два события типа MagazineListHandler

* **MagazineAdded**, которое происходит при добавлении элемента в коллекциию; cобытие передает через объект MagazineListHandlerEventArgs имя коллекции, строку с информацией о том, что в коллекцию был добавлен элемент, и номер добавленного элемента в списке List<Magazine>;
* **MagazineReplaced**, которое происходит, когда одной из ссылок, входящих в коллекцию, присваивается новое значение; событие передает через объект MagazineListHandlerEventArgs имя коллекции, строку с информацией о том, что в коллекции был заменен элемент, и номер замененного элемента.

Событие MagazineAdded бросают методы класса MagazineCollection

* AddDefaults();
* AddMagazines (params Magazine[] ) ;

Событие MagazineReplaced бросают

* метод Replace (int j, Magazine mg);
* метод set индексатора, определенного в классе MagazineCollection.

Определить класс **Listener** для накопления информации об изменениях в коллекциях MagazineCollection. В классе Listener информация хранится в списке из элементов типа **ListEntry,** каждый элемент списка содержит информацию об отдельном изменении в коллекции MagazineCollection.

Класс **ListEntry** содержит:

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о том, какое событие произошло в коллекции;
* номер добавленного или измененного элемента;
* конструктор для инициализации полей класса;
* перегруженную версию метода string ToString().

Класс **Listener** содержит

* список изменений System.Collections.Generics.List<ListEntry>;
* обработчик событий MagazineAdded и MagazineReplaced, который на основе информации из объекта MagazineListHandlerEventArgs, создает элемент ListEntry и добавляет его в список изменений;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<ListEntry>.

В методе **Main()**

1. Создать две коллекции MagazineCollection.
2. Создать два объекта типа Listener, один объект Listener подписать на события MagazineAdded и MagazineReplaced из первой коллекции MagazineCollection, другой объект Listener подписать на события MagazineAdded из обеих коллекций MagazineCollection.
3. Внести изменения в коллекции MagazineCollection

* добавить элементы в коллекции;
* заменить некоторые элементы из коллекций с помощью метода Replасе класса MagazineCollection;
* присвоить некоторым элементам коллекций новые значения c помощью индексатора класса MagazineCollection.

1. Вывести данные обоих объектов Listener.

## Вариант 3. Требования к программе

Определить новую версию класса **ResearchTeamCollection**, которая с помощью событий сообщает об изменениях в коллекции.

Для событий определить делегат **TeamListHandler** с сигнатурой:

**void TeamListHandler**

**(object source, TeamListHandlerEventArgs args);**

Класс **TeamListHandlerEventArgs**, производный от класса System.EventArgs, содержит:

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство типа int с номером элемента, который был добавлен или заменен;
* конструкторы для инициализации класса;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех полях класса.

В новую версию класса **ResearchTeamCollection** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод void InsertAt (int j, ResearchTeam rt), который вставляет элемент rt в список List<ResearchTeam> перед элементом с номером j; если в списке нет элемента с номером j, метод добавляет элемент в конец списка;
* индексатор типа ResearchTeam (с методами get и set) с целочисленным индексом для доступа к элементу списка List<ResearchTeam> с заданным номером.

В новую версию класса **ResearchTeamCollection** добавить два события типа TeamListHandler

* **ResearchTeamAdded**, которое происходит при добавлении элемента в конец списка List<ResearchTeam>; cобытие передает через объект TeamListHandlerEventArgs имя коллекции, строку с информацией о том, что в коллекцию был добавлен элемент, и номер добавленного элемента в списке List<ResearchTeam>;
* **ResearchTeamInserted**, которое происходит, когда новый элемент вставляется перед одним из элементов списка List<ResearchTeam>; событие передает через объект TeamListHandlerEventArgs имя коллекции, строку с информацией о том, что в коллекцию был вставлен элемент, и номер нового элемента.

Событие ResearchTeamAdded бросают методы класса ResearchTeamCollection

* AddDefaults();
* AddResearchTeams ( params ResearchTeam []);
* InsertAt (int j, ResearchTeam rt), если элемента с номером j нет в списке.

Событие ResearchTeamInserted бросает метод InsertAt (int j, ResearchTeam rt), если элемент с номером j есть в списке List<ResearchTeam>.

Определить класс **TeamsJournal** для накопления информации об изменениях в коллекциях типа ResearchTeamCollection. В классе TeamsJournal информация хранится в списке из элементов типа **TeamsJournalEntry,** каждый элемент списка содержит информацию об отдельном изменении в коллекции ResearchTeamCollection.

Класс **TeamsJournalEntry** содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о том, какое событие произошло в коллекции;
* номер нового элемента;
* конструктор для инициализации полей класса;
* перегруженную версию метода string ToString().

Класс **TeamsJournal** содержит

* закрытое поле List<TeamsJournalEntry> для списка изменений;
* обработчик событий ResearchTeamAdded и ResearchTeamInserted. Обработчик использует информацию, которая передается ему через объект TeamListHandlerEventArgs, создает элемент TeamsJournalEntry и добавляет его в список List<TeamsJournalEntry>.
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<TeamsJournalEntry>.

В методе **Main()**

* 1. Создать две коллекции ResearchTeamCollection.
  2. Создать два объекта типа TeamsJournal, один объект TeamsJournal подписать на события ResearchTeamAdded и ResearchTeamInserted из первой коллекции ResearchTeamCollection, другой объект TeamsJournal подписать на события ResearchTeamInserted из обеих коллекций ResearchTeamCollection.
  3. Внести изменения в коллекции ResearchTeamCollection
* добавить элементы в коллекции;
* с помощью метода InsertAt (int j, ResearchTeam rt) перед элементом с номером j, который есть в коллекции, вставить новый элемент;
* вызвать метод InsertAt (int j, ResearchTeam rt) с номером j, которого нет в коллекции.
  1. Вывести данные обоих объектов TeamsJournal.

## Дополнительная работа №8 (10 баллов)

**Делегаты. События** Повышенный уровень сложности

**Информация для всех вариантов**

В работе требуется определить класс, содержащий типизированную коллекцию, который с помощью событий извещает об изменениях в коллекции и в данных ее элементов.

Коллекция состоит из объектов ссылочных типов. Набор ссылок, образующих коллекцию, изменяется при удалении/добавлении элементов или при изменении одной из входящих в коллекцию ссылок, например, когда одной из ссылок присваивается новое значение.

Класс, содержащий коллекцию, сообщает об этих изменениях в коллекции с помощью событий, для которых определяется пользовательский делегат.

Данные элементов, входящих в коллекцию, изменяются при вызове метода get в свойствах, определенных в элементах коллекции. При этом значение самой ссылки, которая входит в коллекцию, не изменяется.

Элементы сообщают об изменениях в своих данных с помощью события **PropertyChange** интерфейса **System.ComponentModel.INotifyPropertyChanged**. Делегат, отвечающий этому событию,

**delegate void PropertyChangedEventHandler**

**(Object sender,PropertyChangedEventArgs e);**

определен в базовой библиотеке. В классе **PropertyChangedEventArgs** есть конструктор с единственным параметром типа string, через который обработчикам события обычно передается имя свойства, которое является источником изменений данных объекта.

Класс, содержащий коллекцию, должен подписаться на событие PropertyChanged для каждого элемента коллекции. Подписку на событие PropertyChanged надо выполнить во всех методах, которые добавляют в коллекцию новые элементы. При удалении элемента из коллекции от подписки на его событие PropertyChanged необходимо отказаться.

При подписке на событие PropertyChanged как обработчик события надо использовать метод класса с коллекцией. В этом методе события PropertyChanged, которые происходят в элементах коллекции, преобразуются в события класса, содержащего коллекцию. Для этого из информации, которую получает обработчик события PropertyChanged, формируются данные для нового события, которое бросается из класса-коллекции.

События регистрируются в специальных классах-слушателях. Класс-слушатель должен отслеживать изменения только в тех объектах, которые входят в коллекцию.

## Вариант 1.

Определить новые версии классов **Student** и **StudentCollection<TKey>**

Новая версия класса **Student** реализует интерфейс **System.ComponentModel.INotifyPropertyChanged**. Событие **PropertyChanged** из интерфейса System.ComponentModel.INotifyPropertyChanged происходит при изменении значений свойств класса **Student**, связанных с номером группы и формой обучения. Название свойства, значение которого изменилось, событие PropertyChanged передает своим обработчикам через свойство PropertyName класса PropertyChangedEventArgs.

Для информации о типе изменений, которые произошли в коллекциях, определить перечисление (enum) **Action** со значениями **Add**, **Remove** и **Property**.

Для события, которое бросают методы класса StudentCollection<TKey>, определить делегат **StudentsChangedHandler<TKey>** с сигнатурой:

**void StudentsChangedHandler<TKey>**

**(object source, StudentsChangedEventArgs<TKey> args);**

Класс **StudentsChangedEventArgs<TKey>**, производный от класса System.EventArgs, содержит

* открытое автореализуемое свойство типа string с названием коллекции;
* открытое автореализуемое свойство типа Action с информацией о том, чем вызвано событие, – удалением элемента, добавлением элемента или изменением данных элемента;
* открытое автореализуемое свойство типа string с названием свойства класса Student, которое является источником изменения данных элемента; для событий, брошенных при удалении или добавлении элемента, значение свойства – пустая строка;
* открытое автореализуемое свойство типа TKey с ключом добавленного, удаленного или измененного элемента;
* конструктор c параметрами типа string, Action, string и TKey для инициализации значений всех свойств класса;
* перегруженную версию метода string ToString().

В новую версию класса **StudentCollection<TKey>** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Remove(Student st) для удаления элемента со значением st из словаря Dictionary<TKey, Student>; если в словаре нет элемента st, метод возвращает значение false;
* событие **StudentsChanged** типа **StudentsChangedHandler<TKey>**, которое происходит, когда в коллекцию добавляются элементы, из нее удаляется элемент или изменяются данные одного из ее элементов.

Определить класс **Journal**, который можно использовать для накопления информации об изменениях в коллекциях типа StudentCollection<TKey>. Класс Journal хранит информацию об изменениях в коллекциях в списке объектов типа **JournalEntry**. Класс JournalEntry содержит информацию об отдельном изменении, которое произошло в коллекциях.

Класс **JournalEntry** содержит автореализуемые свойства

* типа string с названием коллекции;
* типа Action c информацией о типе события;
* типа string с названием свойства класса Student, которое явилось причиной изменения данных элемента;
* типа string с текстовым представлением ключа добавленного, удаленного или измененного элемента;
* конструктор для инициализации всех свойств класса;
* перегруженную версию методаstring ToString().

Класс **Journal** содержит

* закрытое поле типа System.Collections.Generic.List<JournalEntry>;
* обработчик события StudentsChanged, который на основе информации из объекта StudentsChangedEventArgs, создает элемент JournalEntry и добавляет его в список List<JournalEntry>;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<JournalEntry>.

В методе **Main()**

* 1. Создать две коллекции StudentCollection<string> с разными названиями.
  2. Создать объект Journal и подписать его на события StudentsChanged из обеих коллекций StudentCollection<string>.
  3. Внести изменения в коллекции StudentCollection<string>:
* добавить элементы Student в коллекции;
* изменить значения разных свойств элементов, входящих в коллекцию;
* удалить элемент из коллекции;
* изменить данные в удаленном элементе.

4. Вывести данные объекта Journal.

## Вариант 2.

Определить новые версии классов **Edition, Magazine** и **MagazineCollection<TKey>.** Новая версия класса **Edition** реализует интерфейс **System.ComponentModel.INotifyPropertyChanged**. Событие **PropertyChanged** из интерфейса System.ComponentModel.INotifyPropertyChanged происходит при изменении значений свойств класса **Edition**, связанных с тиражом и датой выхода издания. Название свойства, значение которого изменилось, событие PropertyChanged передает своим обработчикам через свойство PropertyName класса PropertyChangedEventArgs.

Для информации о типе изменений, которые произошли в коллекциях, определить перечисление (enum) **Update** со значениями **Add**, **Replace** и **Property**.

Для события, которое бросают методы новой версии класса MagazineCollection<TKey>, определить универсальный делегат **MagazinesChangedHandler<TKey>** с сигнатурой:

**void MagazinesChangedHandler<TKey>**

**(object source, MagazinesChangedEventArgs<TKey> args);**

Класс **MagazinesChangedEventArgs<TKey>**, производный от класса System.EventArgs, содержит

* открытое автореализуемое свойство типа string с названием коллекции;
* открытое автореализуемое свойство типа Update с информацией о том, чем вызвано событие, – добавлением нового элемента в коллекцию, заменой элемента в коллекции или изменением данных элемента;
* открытое автореализуемое свойство типа string с названием свойства класса Magazine, которое является источником изменения данных элемента; для событий, порожденных добавлением или заменой элемента, значение свойства – пустая строка;
* открытое автореализуемое свойство типа TKey с ключом элемента, который был добавлен в коллекцию, заменил один из элементов коллекции или элемента, у которого были изменены данные;
* конструктор c параметрами типа string, Update, string и TKey для инициализации значений всех свойств класса;
* перегруженную версию метода string ToString().

В новую версию класса **MagazineCollection<TKey>** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Replace(Magazine mold, Magazine mnew) для замены в словаре Dictionary<TKey, Magazine> элемента со значением mold на элемент со значением mnew; если в словаре нет элемента со значением mold, метод возвращает значение false;
* событие **MagazinesChanged** типа MagazinesChangedHandler<TKey>, которое происходит при добавлении нового элемента в коллекцию, замене элемента в коллекции или при изменении данных одного из ее элементов.

Определить класс **Listener**, собирающий информацию об изменениях в классе MagazineCollection<TKey>. Класс Listener содержит список из элементов типа **ListEntry**. Каждый элемент ListEntry содержит информацию об отдельном изменении объекта MagazineCollection<TKey>, в результате которого произошло событие MagazinesChanged.

Класс **ListEntry** содержит автореализуемые свойства

* типа string с названием коллекции;
* типа Update c информацией о типе события;
* типа string с названием свойства класса Magazine, которое явилось причиной изменения данных элемента;
* типа string с текстовым представлением ключа добавленного, удаленного или измененного элемента;
* конструктор для инициализации всех свойств класса;
* перегруженную версию метода string ToString().

Класс **Listener** содержит

* закрытое поле типа System.Collections.Generics.List<ListEntry>;
* обработчик события MagazinesChanged, который на основе информации из объекта MagazinesChangedEventArgs, создает элемент ListEntry и добавляет его к списку;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<ListEntry>.

В методе **Main()**

1. Создать два объекта MagazineCollection<string> с разными названиями.

2. Создать объект типа Listener и подписать его на события MagazinesChanged из обоих объектов MagazineCollection<string>.

3. Внести изменения в MagazineCollection<string>:

* добавить элементы в коллекции;
* изменить значения разных свойств элементов, входящих в коллекцию;
* заменить один из элементов коллекции;
* изменить данные в элементе, который был удален из коллекции при замене элемента.

4. Вывести данные объекта Listener.

## Вариант 3.

Определить новые версии классов **ResearchTeam** и **ResearchTeamCollection<TKey>** .

Новая версия класса **ResearchTeam** реализует интерфейс **System.ComponentModel.INotifyPropertyChanged**. Событие **PropertyChanged** из интерфейса System.ComponentModel.INotifyPropertyChanged происходит при изменении значений свойств класса **ResearchTeam**, связанных с названием темы и продолжительностью исследований. Название свойства, значение которого изменилось, событие PropertyChanged передает своим обработчикам через свойство PropertyName класса PropertyChangedEventArgs.

Для информации о типе изменений, которые произошли в коллекциях, определить перечисление (enum) **Revision** со значениями **Remove, Replace** и **Property**.

Для события **ResearchTeamsChanged**, которое бросают методы новой версии класса ResearchTeamCollection<TKey>, определить универсальный делегат **ResearchTeamsChangedHandler<TKey>** с сигнатурой:

**void ResearchTeamsChangedHandler<TKey>**

**(object source, ResearchTeamsChangedEventArgs<TKey> args);**

Класс **ResearchTeamsChangedEventArgs<TKey>**, производный от класса System.EventArgs, содержит

* открытое автореализуемое свойство типа string с названием коллекции;
* открытое автореализуемое свойство типа Revision с информацией о том, чем вызвано событие, – удалением, заменой элемента или изменением данных элемента;
* открытое автореализуемое свойство типа string с названием свойства класса ResearchTeam, которое является источником изменения данных элемента; для событий, брошенных при удалении или замене элемента, значение свойства – пустая строка;
* открытое автореализуемое свойство типа int c номером регистрации объекта ResearchTeam для элемента, который был удален, в том числе и при замене элемента, или элемента, данные которого были изменены;
* конструктор c параметрами типа string, Revision, string и int для инициализации значений всех свойств класса;
* перегруженную версию метода string ToString().

В новую версию класса **ResearchTeamCollection<TKey>** добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Remove(ResearchTeam rt) для удаления элемента со значением rt из словаря Dictionary<TKey, ResearchTeam>; если в словаре нет элемента rt, метод возвращает значение false;
* метод bool Replace(ResearchTeam rtold, ResearchTeam rtnew) для замены в словаре Dictionary<TKey, ResearchTeam > элемента со значением rtold на элемент со значением rtnew; если в словаре нет элемента со значением rtold, метод возвращает значение false;
* событие **ResearchTeamsChanged** типа ResearchTeamsChangedHandler<TKey>, которое происходит, когда изменяется набор элементов в коллекции-словаре Dictionary<TKey,ResearchTeam> или изменяются данные одного из ее элементов.

Определить класс **TeamsJournal**, который можно использовать для накопления информации об изменениях в коллекциях типа ResearchTeamCollection<TKey>. Класс TeamsJournal содержит список из элементов типа **TeamsJournalEntry**. Каждый элемент TeamsJournalEntry содержит информацию об отдельном изменении объекта ResearchTeamCollection<TKey>, в результате которого произошло событие ResearchTeamsChanged.

Класс **TeamsJournalEntry** содержит автореализуемые свойства

* типа string с названием коллекции;
* типа Revision c информацией о типе события;
* типа string с названием свойства класса ResearchTeam, которое явилось причиной изменения данных элемента;
* типа int с номером регистрации объекта ResearchTeam для удаленного элемента или элемента, данные которого были изменены;
* конструктор для инициализации всех свойств класса;
* перегруженную версию метода string ToString().

Класс **TeamsJournal** содержит

* закрытое поле List<TeamsJournalEntry> для списка изменений;
* обработчик события ResearchTeamsChanged; обработчик использует информацию, которая передается ему через объект ResearchTeamsChangedEventArgs, создает элемент TeamJournalEntry и добавляет его к списку List<TeamsJournalEntry>;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах списка List<TeamsJournalEntry>.

В методе **Main()**

1. Создать две коллекции ResearchTeamCollection<string>.

2. Создать объект TeamsJournal, подписать его на события ResearchTeamsChanged из обоих объектов ResearchTeamCollection<string>.

3. Внести изменения в коллекции ResearchTeamCollection<string>

* добавить элементы в коллекции;
* изменить значения разных свойств элементов, входящих в коллекцию;
* удалить элемент из коллекции;
* изменить данные в удаленном элементе;
* заменить один из элементов коллекции;
* изменить данные в элементе, который был удален из коллекции при замене элемента.

4. Вывести данные объекта TeamsJournal.

# Практическая работа №9. Система ввода-вывода

Цель: файловая система ввода- вывода, классы для работы с файлами.

**Файл** - именованная информация на внешнем носителе (на жестком или гибком магнитном диске). Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатура и принтер, также можно рассматривать как частные случаи файлов.

Передача данных с внешнего устройства в оперативную память называется чтением, или вводом, обратный процесс –записью, или выводом.

Чтение-запись в языке C# выполняется с помощью подсистемы ввода-вывода и классов библиотеки .NET.

**Поток** (stream)– это любой перенос данных от источника к приемнику. Потоки обеспечивают работу, как со стандартными так и с определенными пользователем типами данных. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производиться обмен. Обмен с потоком для повышения скорости передачи данных производится, как правило, через специальную область оперативной памяти – **буфер**. Буфер выделяется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не заполниться. Только после этого и после специальной команды сброса происходит передача данных на внешнее устройство.

Для поддержки потоков библиотека .NET содержит иерархию классов, которые определены в пространстве имен **System.IO.**

Подробная информация находится в лекции «Система ввода-вывода»

# Задание на практическую работу №9 (10 баллов)

Определить новые версии классов **Student** (вариант **1**), **Magazine** (вариант **2**) и **ResearchTeam** (вариант **3**). В сформулированных ниже требованиях для этих классов использовано общее обозначение **T**.

В новые версии классов добавить **экземплярные** методы

* T DeepCopy() для создания полной копии объекта;
* bool Save(string filename) для сохранения данных объекта в файле;
* bool Load(string filename) для инициализации объекта данными из файла;
* bool AddFromConsole() для добавления в один из списков класса нового элемента, данные для которого вводятся с консоли;

и статические методы

* static bool Save(string filename, T obj) для сохранения объекта в файле;
* static bool Load(string filename, T obj) для восстановления объекта из файла;

В экземплярном методе **T DeepCopy()**возвращается полная копия исходного объекта.

Экземплярный метод **bool Save(string filename)** сохраняет все данные вызывающего объекта в файл с именем filename. Если файл с именем filename существует, приложение его перезаписывает. Если такого файла нет, приложение его создает. Метод возвращает значение true, если сохранение завершилась успешно, и значение false в противном случае.

Экземплярный метод **bool Load(string filename)** считывает данные из файла с именем filename и использует их для инициализации вызывающего объекта. Метод возвращает значение true, если инициализация завершилась успешно. Если полностью выполнить инициализацию объекта не удалось, исходные данные объекта должны остаться без изменения. В этом случае метод возвращает значение false.

Статические методы **bool Save(string filename, T obj)** и **bool Load(string filename, T obj)** получают через параметры имя файла и ссылку на объект, для которого выполняется сохранение или инициализация. Методы возвращают значение true, если сохранение /инициализация завершилась успешно, и значение false в противном случае. Если полностью выполнить инициализацию объекта не удалось, исходные данные объекта должны остаться без изменения.

Во всех реализациях методов сохранения/восстановления данных из файла, операции открытия файла должны находиться в блоках try-catch-finally.

В методе **bool AddFromConsole()** для добавления нового элемента в один из списков класса T

* пользователь получает приглашение ввести данные в виде одной строки символов с разделителями; приглашение содержит описание формата строки ввода, в том числе информацию о том, какие символы можно использовать в качестве разделителей;
* выполняется разбор данных; операции преобразования данных, которые могут бросить исключение, должны находиться в блоке try-catch;
* если разбор введенных данных был завершен успешно, в список добавляется новый элемент и метод возвращает значение true; в противном случае пользователь получает сообщение о том, что при вводе были допущены ошибки и возвращаемое значение метода равно false.

В **варианте** **1** элементы, данные для которых вводятся с консоли, добавляются в список экзаменов System.Collections.Generic.List<Exam>. Вводятся название предмета, оценка и дата экзамена.

В **варианте 2** элементы добавляются в список статей в журнале System.Collections.Generic.List<Article>. Вводятся название статьи, данные автора статьи для объекта типа Person и рейтинг статьи.

В **варианте** **3** элементы добавляются в список публикаций System.Collections.Generic.List<Paper>. Вводятся название публикации, данные автора статьи для объекта типа Person и дата публикации.

В методе **Main()**

1. Создать объект типа T с непустым списком элементов, для которого предусмотрен ввод данных с консоли. Создать полную копию объекта с помощью метода, и вывести исходный объект и его копию.

2. Предложить пользователю ввести имя файла:

* если файла с введенным именем нет, приложение должно сообщить об этом и создать файл;
* если файл существует, вызвать метод Load(string filename) для инициализации объекта T данными из файла.

3. Вывести объект T.

4. Для этого же объекта T сначала вызвать метод AddFromConsole(), затем метод Save(string filename). Вывести объект T.

5. Вызвать последовательно

* статический метод Load( string filename, T obj), передав как параметры ссылку на тот же самый объект T и введенное ранее имя файла;
* метод AddFromConsole();
* статический метод Save (string filename, T obj).

6. Вывести объект T.

Приложение должно работать в режиме накопления. Если выбирается один и тот же файл для записи, и пользователь вводит данные без ошибок, при каждом следующем выполнении приложения к списку добавляются два новых элемента. Приложение должно обрабатывать все исключения, которые могут возникнуть из-за ошибок при вводе данных. Независимо от того, корректно были введены данные или при вводе были допущены ошибки, все файловые потоки должны быть закрыты.

7. Для папки с сохраненными файлами рекурсивно выдать список её файлов и подпапок.

**Список литературы**

Эндрю Троелсен ЯЗЫК ПРОГРАММИРОВАНИЯ C#5.0 И ПЛАТФОРМА .NET 4.5 6-е изд.: Пер. с англ.- М. : ООО «И.Д.Вильямс».2013.-1312с.