



Обучающий курс

C#

(junior специалист)

<https://job.dex-it.ru>



Оглавление

Введение	5
Рабочий процесс Git	6
Базовые принципы ООП	8
1. Наследование	9
2. Инкапсуляция	9
3. Полиморфизм	10
Типы значений и ссылочные типы (Value type and Reference type)	11
1. Составные типы	14
2. Копирование значений	15
3. Ссылочные типы внутри типов значений	15
4. Объекты классов как параметры методов	16
Приведение и преобразование типов	20
1. Неявные преобразования	20
2. Явные преобразования	21
Упаковка и распаковка	24
Делегаты функций. Анонимные методы. Лямбда выражения	27
1. Делегаты функций	27
2. Анонимные методы	28
3. Лямбда выражения	28
События. Events	30
List, Dictionary	32
Эквивалентность, Equals, GetHashCode.	35
1. Эквивалентность - оператор “==”	35
2. Метод Equals	37
3. Метод GetHashCode	38
4. Equals и GetHashCode применительно к структурам	39
Comparable - сравнение объектов	43
Обработка исключений (Exception handling)	44
Базовые интерфейсы для работы со списками (IEnumerable, IEnumerator)	50

Generic Type, Generic Member	54
Работа с базой данных. Проектирование, нормализация, DataGrip	59
1. Создание новой базы	59
2. Создание новой таблицы	60
3. Нормализация	61
SQL практика	64
Введение в EntityFramework, миграции	66
1. Fluent API	69
2. Аннотации	70
3. Миграции	71
IDisposable. Ключевое слово using	75
Stream. FileStream	85
1. Работа с FileStream	86
2. Работа с StreamReader/StreamWriter	90
3. Работа с CSV файлами	91
Расширения, Extensions	95
Reflection - рефлексия, метаданные классов	97
Serialization (Json, XML, Binary)	99
1. Бинарная сериализация	100
2. Сериализация в XML	101
3. Сериализация в JSON	102
Asp.net controller	104
Класс HttpClient и отправка запросов	108
1. Основные операции по HTTP	108
2. Путь/маршрут (route/path), URI	108
3. Методы HttpClient	109
4. Отправка и получение данных	110
5. Код Состояния	115
Регулярные выражения (Regular expression)	117
Многопоточность, блокировки, дедлоки	120
Задачи и класс Task	127

1. Ожидание задачи	128
2. Вложенные задачи	129
3. Отмена задач и параллельных операций. CancellationToken	132
Асинхронное программирование. Асинхронные методы, async и await	135
Последовательный и параллельный вызов асинхронных операций	137
Модульное тестирование	140

Введение

Цель - дать теоретическую и практическую основу начинающим разработчикам в стеке C# .Net. Уровень - конспект рассчитан на новичков, но обладающих знаниями о примитивных языковых конструкциях (if..else, for, foreach...).

Требования - для прохождения и усвоения материала необходимо установить следующие инструменты

1. [Visual Studio](#) или [Rider](#) (лицензия на Rider для студентов - [бесплатно](#))
2. Скачать [примеры кода](#)
3. Научиться запускать тестовые примеры (выполнены в виде NUnit тестов)
 - a. для Visual Studio (от 15.9.x) - нужно открыть "Обозреватель тестов" и собрать решение (build solution)
 - b. для среды Rider - каждый тест подсвечивается на полях или Alt+8 TestExplore
4. Успехов!

Рабочий процесс Git

Последовательность действий при работе с git:

1. Перед началом работы, на сервере Github, делаем удаленную копию предложенного преподавателем репозитория, при помощи команды «Fork». При этом к имени репозитория добавляем постфикс-свою фамилию.
2. В рамках своей копии репозитория, из ветки master создаем ветку develop.
3. Из ветки develop в ходе работы, создаем ветки feature (Имя ветки feature/<тема_задача>, например: feature/List_Dictionary_creatureFakeDataService)
 - a. В ветку задачи разработчик может коммитить что угодно
 - b. Перед тем как пушить проверяем список изменений которые мы собрались пушить. Точно ли они соответствуют нашим ожиданиям?
 - c. При переключении с ветки на ветку, пушим на сервер
4. Когда работа над веткой feature завершена, она сливается с веткой develop, с помощью merge request(MR)/pull request(PR). При открытии MR добавляем преподавателя в качестве рецензента.

Особенности MR:

- Код попадаемый в MR должен быть рабочим. Нужно проверить: тесты успешно отрабатывают, решение успешно строится(build).
- Перед созданием MR feature - обновляем свою ветку до последней версии origin/develop(делаем merge dev ветки в свою), и решаем конфликты
- Если ваш MR не прошел успешно review, то его закрывают. После доработки, закрытый MR открываем

Комментарий commit'a:

Шаблон комментария: <task_number> <type>: <title>

- <type> – указываем тип изменения
- <title> – указываем вкратце(до 50 символов) что изменяет, для чего исправление. Заголовок указывается со строчной буквы, в повелительном наклонении, точка в конце заголовка не ставится.

Список типов изменений <type>:

- feat: добавление нового функционала
- fix: исправление какой-либо программной ошибки

Ресурсы:

[Git How To: Guided Git Tutorial](#)

[GitHub Flow / Хабр \(habr.com\)](#)

Базовые принципы ООП

Основные принципы - наследование, инкапсуляция и полиморфизм. С# - полностью поддерживает все принципы ООП, единственное ограничение - отсутствие множественного наследования (наследовать можно только 1 класс).

Кратко рассмотрим каждый принцип на примере:

```
public abstract class BaseLetter
{
    public string SendTo { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }

    public void Send()
    {
        var body = CreateLetterBody();
        Save(body);
        Sending(body);
    }

    protected abstract object CreateLetterBody();

    private void Sending(object body) {...}

    private void Save(object body) {...}
}
```

```
public class Letter : BaseLetter
{
    protected override object CreateLetterBody()
    {
        return new object(); // создаем тело сообщения
    }
}
```



```
public class SecureLetter : Letter
{
    protected override object CreateLetterBody()
    {
        var body = base.CreateLetterBody();
        var encBody = Encrypt(body);
        return encBody;
    }

    private object Encrypt(object body)
    {
        return body; // шифруем тело сообщения
    }
}
```

1. Наследование

Основное предназначение - повторное использование кода путем выделения общих частей и вынесения их в абстракции. Базовый класс писем BaseLetter содержит общую логику отправки писем - метод Send, а вот классы Letter и SecureLetter отличаются только реализацией методов CreateLetterBody, в то время как базовая абстракция BaseLetter вообще не содержит реализации такого метода.

2. Инкапсуляция

Принцип говорит о том что логически связанные данные и методы рассматриваются в системе как единое целое - класс. В нашем примере это свойства SendTo, Author, Title и метод Send() и другие. Таким образом для работы с ними другим членам класса не нужно передавать их в параметры, как например это было бы в процедурном подходе. С этим понятием еще тесно связано понятие область видимости - private, protected, internal, public. Это модификаторы доступа, они определяют кому будет виден член класса, всем (public), только членам класса (private), только наследникам и членам класса (protected), internal - разберите сами. Это очень важный компонент в проектировании класса, если выразить кратко, то внутреннее устройство класса д.б. скрыто, а пользователи видели снаружи только необходимый набор членов класса.

3. Полиморфизм

Принцип позволяет работать с объектами разных типов через общего родителя.

Пример:

```
var letters = new BaseLetter[] {new Letter(), new SecureLetter()};

foreach (var letter in letters)
{
    letter.Send(); // отправляем письма
}
```

причем для каждого типа вызовется его собственная реализация CreateLetterBody().

Самостоятельно:

- изучить интерфейсы, область применения
- какие из принципов ООП применимы к статическим членам класса
- реализовать несколько классов, с применением всех принципов ООП, продемонстрировать в приложении (консоль). Области для примера: животные, автомашины, геометрические фигуры, астрономические объекты.

[Полный пример](#)

Ресурсы:

[ООП на простых примерах](#)

[Инкапсуляция](#)

[Наследование](#)

[Полиморфизм](#)

Типы значений и ссылочные типы (Value type and Reference type)

В C# существуют две разновидности типов: типы значений и ссылочные типы. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров `in`, `ref` и `out`; см. описание модификатора параметров [in](#), [ref](#) и [out](#)). В переменных ссылочных типов хранятся ссылки на их данные (объекты), а переменные типа значений содержат свои данные непосредственно. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому операции над одной переменной могут затрагивать объект, на который ссылается другая переменная.

Типы значений:

- Целочисленные типы (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`)
- Типы с плавающей запятой (`float`, `double`)
- Тип `decimal`
- Тип `bool`
- Тип `char`
- Перечисления `enum`
- Структуры (`struct`)

Ссылочные типы:

- Тип `object`
- Тип `string`
- Классы (`class`)
- Интерфейсы (`interface`)
- Делегаты (`delegate`)

В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (`heap`). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего (аналогично стопке тарелок на кухне). Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове

каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Подробнее о расположении значений и объектов в стеке и куче можно ознакомиться в следующей [статье](#).

Рассмотрим пример простой программы:

```
using System;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            Calculate(5);
            Console.ReadKey();
        }

        static void Calculate(int t)
        {
            int x = 6;
            int y = 7;
            int z = y + t;
        }
    }
}
```

При запуске такой программы и вызове метода `Calculate` в стек будут помещены значения `t`, `x`, `y` и `z`. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

Если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредственное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода `Calculate` представляют значимый тип - тип `int`, поэтому в стеке будут храниться их числовые значения.

Ссылочные типы хранятся в куче или `heap`, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

Рассмотрим пример простой программы:

```
using System;

namespace Examples
{
    class Program
    {
        static void Main(string[] args)
        {
            Calculate(5);
            Console.ReadKey();
        }

        static void Calculate(int t)
        {
            int x = 6;
            int y = 7;
            int z = y + t;
        }
    }
}
```

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (heap). А сам объект размещается в кучу и доступен по адресу, значение которого сохранено для ссылочной переменной в стеке. Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в heap нет больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Так, в частности, если мы изменим метод Calculate следующим образом:

```
static void Calculate(int t)
{
    object x = 6;
    int y = 7;
    int z = y + t;
}
```

Теперь переменная x будет хранить не значение 6, а значение адреса в памяти (в куче), где записано само значение 6, так как она представляет ссылочный тип object.

1. Составные типы

Теперь рассмотрим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

```
using System;
namespace Examples
{
    class Program
    {
        private static void Main(string[] args)
        {
            State state1 = new State(); // State - структура,
            //ее данные размещены в стеке
            Country country1 = new Country(); // Country - класс,
            //в стек помещается ссылка на адрес в куче,
            //а в куче располагаются все данные
        }
    }

    struct State
    {
        public int x;
        public int y;
        public Country country;
    }

    class Country
    {
        public int x;
        public int y;
    }
}
```

Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в куче (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в куче, по которому размещен данный объект. Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в куче.

Однако в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?

```

...
private static void Main(string[] args)
{
    State state1 = new State();
    state1.country = new Country();
    Country country1 = new Country();
}
...

```

Значение переменной `state1.country` также будет храниться в куче, так как эта переменная представляет ссылочный тип.

2. Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а значение ссылки на этот объект в куче. Рассмотрим следующий пример:

Так как `state1` - структура, то при присвоении `state1 = state2` она получает копию структуры `state2`. А объект класса `country1` при присвоении `country1 = country2;` получает ссылку на тот же объект, на который указывает `country2`. Поэтому с изменением `country2`, так же будет меняться и `country1`.

3. Ссылочные типы внутри типов значений

Теперь рассмотрим еще раз пример, когда внутри структуры у нас может быть переменная ссылочного типа, например, какого-нибудь класса:

```

class Program
{
    static void Main(string[] args)
    {
        State state1 = new State(); // Структура State

        State state2 = new State();

        state2.x = 1;
        state2.y = 2;
        state1 = state2;
        state2.x = 5; // state1.x=1 по-прежнему
    }
}

```

```

        Console.WriteLine(state1.x); // 1
        Console.WriteLine(state2.x); // 5

        Country country1 = new Country(); // Класс Country

        Country country2 = new Country();

        country2.x = 1;
        country2.y = 4;
        country1 = country2;
        country2.x = 7; // теперь и country1.x = 7,
        //так как обе ссылки и country1
        //и country2 указывают на один объект в куче

        Console.WriteLine(country1.x); // 7
        Console.WriteLine(country2.x); // 7

        Console.Read();
    }

```

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в куче. И при присвоении двух структур `state1 = state2`; структура `state1` также получит ссылку на объект `country` в куче. Поэтому изменение `state2.country` повлечет за собой также изменение `state1.country`.

4. Объекты классов как параметры методов

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в куче. И при присвоении двух структур `state1 = state2`; структура `state1` также получит ссылку на объект `country` в куче. Поэтому изменение `state2.country` повлечет за собой также изменение `state1.country`.

```

class Program
{
    static void Main(string[] args)
    {
        State state1 = new State(); // Структура State

        State state2 = new State();

        state2.country = new Country();
        state2.country.x = 5;
        state1 = state2;
        state2.country.x = 8; // теперь и state1.country.x=8,    //так
        как state1.country и state2.country
        //указывают на один объект в куче
    }
}

```



```

        Console.WriteLine(state1.country.x); // 8
        Console.WriteLine(state2.country.x); // 8

        Console.Read();
    }
}

struct State
{
    public int x;
    public int y;
    public Country country;
}

class Country
{
    public int x;
    public int y;
}

```

Организацию объектов в памяти следует учитывать при передаче параметров по значению и по ссылке. Если параметры методов - это ссылочные переменные, указывающие на объекты классов, то использование параметров имеет некоторые особенности. Например, создадим метод, который в качестве параметра принимает переменную, указывающую на объект `Person`:

При передаче объекта класса в качестве аргумента метода передается копия ссылки на объект. Эта копия указывает на тот же объект, что и исходная ссылка, потому мы можем изменить отдельные поля и свойства объекта. Например, строка `person.name = "Alice"` изменяет значение поля объекта с "Tom" на "Alice".

А другая строка `person = new Person { name = "Bill", age = 45 }` создаст новый объект в памяти, и переменная `person` теперь будет указывать на новый объект в памяти. И даже если после этого мы будем изменять объект на который ссылается переменная `person`, то это никак не повлияет на объект, на который ссылается переменная `p` в методе `Main`, поскольку ссылка `p` все еще указывает на старый объект в памяти.

Такая передача называется передачей параметра по значению.

Есть возможность передать параметр по ссылке (с помощью ключевого слова `ref`). В этом случае в метод в качестве аргумента передается сама ссылка на объект в памяти, а не ее копия. Поэтому можно изменить как поля и свойства объекта, так и сам объект, на который ссылается переменная в методе:

```

class Program
{
    static void Main(string[] args)
    {
        Person p = new Person { name = "Tom", age=23 };
        ChangePerson(p);

        Console.WriteLine(p.name); // Alice
        Console.WriteLine(p.age); // 23

        Console.Read();
    }

    static void ChangePerson(Person person)
    {
        // сработает
        person.name = "Alice";
        // сработает только в рамках данного метода
        person = new Person { name = "Bill", age = 45 };
        Console.WriteLine(person.name); // Bill
    }
}

class Person
{
    public string name;
    public int age;
}

```

Операция `new` создает новый объект в памяти, и теперь ссылка `person` (она же ссылка `p` из метода `Main`) будет указывать уже на новый объект в памяти.

Практическая часть:

В папку “Application” новый проект по шаблону библиотека классов, с именем “Models”. В рамках этого проекта, спроектировать и реализовать свои типы “Person” (человек), и дочерние типы “Employee” (сотрудник), “Client” (клиент), структуру “Currency” (валюта).

В папку “Tools” добавить проект (шаблон консольное приложение) “PracticeWithTypes”, в рамках которого реализуем методы:

а) метод, обновляющий контракт сотрудника. Принимает на вход сотрудника и создает контракт (свойство класса “Employee”, строка) на основе его данных. Результат присваивается обратно в тело сотрудника, свойству “Contract”;

б) метод обновляющий сущность валюты. (Метод принимает на вход экземпляр структуры “Currency”, меняет значение ее свойств);

Сравнить результат работы методов и объяснить его.

Вопросы для самоконтроля:

- Есть ли различие по хранению структуры и класса в памяти?
- Что происходит с объектом при передаче его в качестве параметра метода?
- При присваивании одной структуры в другую что копируется?

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/value-types>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/reference-types>
- <https://metanit.com/sharp/tutorial/2.16.php>
- <https://nekliukov.gitbooks.io/konspekt-po-kinge-dzheffri-rihtera-clr-via-c/content/prektirovanie-tipov/primitivi-ssilochnie-i-znachimie-tipi/ssilochnie-i-znachimie-tipi.html>

Приведение и преобразование типов

Иногда может потребоваться скопировать значение в переменную или параметр метода другого типа. Например, может потребоваться передать целочисленную переменную в метод, параметр которого имеет тип `double`. Или может понадобиться присвоить переменную класса переменной типа интерфейса. Такого рода операции называются преобразованиями типа. В C# можно выполнять следующие виды преобразований.

- **Неявные преобразования.** Специальный синтаксис не требуется, так как преобразование является строго типизированным и данные не будут потеряны. Примеры включают преобразования из меньших в большие целочисленные типы и преобразования из производных классов в базовые классы.
- **Явные преобразования (приведения)** . Для явных преобразований требуется оператор приведения. Приведение требуется, если в ходе преобразования данные могут быть утрачены или преобразование может завершиться сбоем по другим причинам. Типичными примерами являются числовое преобразование в тип с меньшей точностью или меньшим диапазоном и преобразование экземпляра базового класса в производный класс.

Существуют также пользовательские преобразования и преобразования с использованием вспомогательных классов, но в рамках данного урока мы их не рассматриваем.

1. Неявные преобразования

Для встроенных числовых типов неявное преобразование можно выполнить, если сохраняемое значение может уместиться в переменной без усечения или округления. При использовании целочисленных типов это означает, что диапазон исходного типа является надлежащим подмножеством диапазона для целевого типа. Например, переменная типа [`long`](#) (64-разрядное целое число) может хранить любое значение, которое может хранить переменная [`int`](#) (32-разрядное целое число). В следующем примере компилятор неявно преобразует значение `intValue` справа в тип `long` перед назначением `longValue`.

Пример неявного преобразования:

```
int intValue = 2147483647;  
long longValue = intValue;
```

Для ссылочных типов неявное преобразование всегда предусмотрено из класса в любой из его базовых классов или интерфейсов. Никакой специальный синтаксис не требуется, поскольку производный класс всегда содержит все члены базового класса.

Пример неявного преобразования:

```
Derived d = new Derived();
Base b = d;

class Base {}
class Derived : Base{}
```

2. Явные преобразования

Если преобразование нельзя выполнить без риска потери данных, компилятор требует выполнения явного преобразования, которое называется приведением. Приведение — это способ явно указать компилятору, что необходимо выполнить преобразование и что вам известно, что может произойти потеря данных. Чтобы выполнить приведение, укажите тип, в который производится приведение, в круглых скобках перед преобразуемым значением или переменной. В следующей программе выполняется приведение типа [double](#) в [int](#). Программа не будет компилироваться без приведения.

Пример приведения типов:

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Приводим double к int.
        a = (int)x; //a=1234
        System.Console.WriteLine(a);
    }
}
```

Для ссылочных типов явное приведение является обязательным, если необходимо преобразовать базовый тип в производный тип:

```
// Создаем объект производного типа
Giraffe g = new Giraffe();

// Неявное преобразование к базовому типу
Animal a = g;

// Приведение базового типа к производному
Giraffe g2 = (Giraffe) a;
```

Операция приведения между ссылочными типами не меняет сам объект, изменяется только "Тип" ссылки на этот объект.

В некоторых преобразованиях ссылочных типов компилятор не может определить, будет ли приведение допустимым. Есть вероятность, что правильно скомпилированная операция приведения завершится сбоем во время выполнения. Приведение типа, завершившееся сбоем во время выполнения, вызывает исключение [InvalidCastException](#). C# предоставляет оператор [is](#), чтобы можно было проверить совместимость перед фактическим выполнением приведения.

Практическая часть:

В папку “Application” добавить новый проект по шаблону библиотека классов, в котором создаем сервис “BankService”, в рамках которого реализуем методы:

а) метод расчета зарплаты владельцев банка = прибыль банка - расходы / количество владельцев (при условии что владелец тоже сущность Employee и ЗП это int)

б) метод преобразования клиента банка в сотрудника, метод принимает на вход сущность “Client” приводит его к типу “Employee” и возвращает его в качестве результата для дальнейшей работы.

Вопросы для самоконтроля:

- Какие виды преобразований типа вы знаете?
- В каких ситуациях допустимо неявное преобразования значимых типов?
- В каких ситуациях допустимо неявное преобразования ссылочных типов?
- В каких ситуациях необходимо приведение значимых типов?
- Для чего нужны операторы (as, is), чем отличаются?

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/casting-and-type-conversions>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/implicit-numeric-conversions-table>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/how-to/safely-cast-using-pattern-matching-is-and-as-operators>

Упаковка и распаковка

Упаковка представляет собой процесс преобразования типа значения в тип `object` или в любой другой тип интерфейса, реализуемый этим типом значения. Когда тип значения упаковывается средой CLR, он инкапсулирует значение внутри экземпляра [System.Object](#) и сохраняет его в управляемой куче. Операция распаковки извлекает тип значения из объекта. Упаковка является неявной; распаковка является явной.

Простыми словами, упаковка и распаковка - это механизм перемещения данных из области стека в кучу - и наоборот.

Помните:

- Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой.
- Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой.

Пример упаковки и распаковки:

```
int i = 123;  
object o = i; // упаковка  
i = (int)o; // распаковка
```

По сравнению с простыми операциями присваивания операции упаковки и распаковки являются весьма затратными процессами с точки зрения вычислений. При выполнении упаковки типа значения необходимо создать и разместить новый объект. Объем вычислений при выполнении операции распаковки, хотя и в меньшей степени, но тоже весьма значителен.

Упаковка используется для хранения типов значений в куче со сбором мусора. Упаковка представляет собой неявное преобразование типа значения в тип `object` или в любой другой тип интерфейса, реализуемый этим типом значения. При упаковке типа значения в куче выделяется экземпляр объекта и выполняется копирование значения в этот новый объект.

В результате упаковки в нашем примере создается ссылка на объект `o` в стеке, которая ссылается на значение типа `int` в куче. Это значение является копией значения типа значения, присвоенного переменной `i`.

Распаковка является явным преобразованием из типа `object` в тип значения или из типа интерфейса в тип значения, реализующего этот интерфейс. Операция распаковки состоит из следующих действий:

- проверка экземпляра объекта на то, что он является упакованным значением заданного типа значения;
- копирование значения из экземпляра в переменную типа значения.

На рисунке ниже представлен результат выполнения нашего кода.

Для успешной распаковки типов значений во время выполнения необходимо, чтобы экземпляр, который распаковывается, был ссылкой на объект, предварительно созданный с помощью упаковки экземпляра этого типа значения. Попытка распаковать `null` создает исключение [NullReferenceException](#). Попытка распаковать ссылку на несовместимый тип значения создает исключение [InvalidCastException](#). В следующем примере показан случай недопустимой распаковки, в результате чего создается исключение `InvalidCastException`. В случае использования `try` и `catch` при возникновении ошибки выводится сообщение.

```
using System;
namespace Examples
{
    class Program
    {
        static void Main()
        {
            int i = 123;
            object o = i; // упаковка

            try
            {
                int j = (short)o; // попытка распаковки
                System.Console.WriteLine("Unboxing OK.");
            }
            catch (System.InvalidCastException e)
            {
                System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
            }
        }
    }
}
```

Вопросы для самоконтроля:

- Что такое упаковка и распаковка?
- В чем ее недостатки?
- Зачем она нужна если у нее такие недостатки?

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/boxing-and-unboxing>
- <https://itvdn.com/ru/blog/article/boxing-unboxing>

Самостоятельно:

- Используя Stopwatch измерить скорость операции упаковка и распаковка

Делегаты функций. Анонимные методы. Лямбда выражения

1. Делегаты функций

Делегат - это объект, который указывает на метод или несколько методов, т.е. **делегат** - это указатель на метод или методы. Имея ссылку на делегат, мы можем добавить, удалить, вызвать методы, которые находятся в делегате.

Для объявления делегата используется ключевое слово `delegate`, после которого идет возвращаемый тип, название и параметры.

Например: `delegate void Message();`

Делегат `Message` в качестве возвращаемого типа имеет тип `void` (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Давайте напишем такие методы:

```
private static void Hello()
{
    Console.WriteLine("Hello");
}

private static void HowAreYou()
{
    Console.WriteLine("How are you?");
}
```

И теперь воспользуемся им при помощи нашего делегата:

```
Message mes; // Создаем переменную делегата
mes = Hello; //Присваиваем адрес метода
mes(); // Вызываем метод
```

Также, делегат может иметь несколько ссылок на разные методы так и на один и тот же.

```
Message mes = Hello;
mes += Hello;
mes += HowAreYou;
mes += Hello;
mes();
```

В результате вызова данного делегата вызовется 2 раза метод Hello затем HowAreYou и снова Hello.

Есть и другой способ вызова делегата, метод Invoke. Если делегат должен принимать параметры то они передаются в метод Invoke:

```
Message mes = Hello;  
mes.Invoke();
```

Делегаты могут быть параметрами функций:

```
private static void ShowMessages(Message mes)  
{  
    mes?.Invoke();  
}
```

2. Анонимные методы

Анонимные методы используются для создания экземпляров делегатов.

```
Message mes = delegate()  
{  
    Console.WriteLine("Hey. I am an anonymous method");  
}
```

3. Лямбда выражения

Лямбда-выражения представляют упрощенную запись анонимных методов.

Лямбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список_параметров) => выражение.

Пример:

```
Message mes = () => Console.WriteLine("Hey. I lambda");
```

Практическая часть:

(объединена с темой List, Dictionary)

Вопросы для самоконтроля:

- Какова роль делегата?
- Всегда ли можно заменить делегат лямбда выражением?

- Чем отличается анонимный метод от делегата?

Самостоятельно:

- Разобрать применение делегатов
(<https://metanit.com/sharp/tutorial/3.43.php>)
- Изучить делегаты Action, Predicate и Func
(<https://metanit.com/sharp/tutorial/3.33.php>)
- Рассмотреть как работает вычитание делегатов.
- События (<https://metanit.com/sharp/tutorial/3.14.php>)
- Перестроить проведённую на практике работу под делегат Func.

События. Events

Конструкция позволяет сигнализировать о происхождении какого-либо события. Так уже повелось на практике, что чаще всего речь идёт об изменении состояния "наблюдаемого" объекта.

Данная конструкция позволяет сообщать информацию заранее неизвестным зависимостям, таким образом достигается ослабление зависимости между сущностями, вместо явной зависимости, где один класс явно ссылается на другой, мы можем сделать данную связь неявной, где один класс сообщает о своём внутреннем состоянии другим, заранее неизвестным классам. Приведем пример класса, который генерирует событие когда внутренний счетчик `_count` достигнет порога `_threshold`.

```
public class Counter
{
    private readonly int _threshold;
    private int _count = 0;

    public event EventHandler ThresholdReached;

    public Counter(int threshold)
    {
        _threshold = threshold;
    }

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    public void Increment()
    {
        _count++;
        if (_count >= _threshold)
        {
            ThresholdReached?.Invoke(this, EventArgs.Empty);
        }
    }
}
```

Теперь о самом важном - подписка и отписка на получение событий.

Основное правило - если мы подписались на событие, то должны обязательно отписаться, когда уничтожаем подписчика, не сделав это мы получаем утечку памяти, потому что "наблюдаемый" хранит все ссылки на "подписчиков".

Пример:

```
var counter = new Counter(10);

EventHandler counterOnThresholdReached = delegate {
    Console.WriteLine("Threshold reached"); };

counter.ThresholdReached += counterOnThresholdReached;

counter.ThresholdReached += (sender, args) => {
    Console.WriteLine("Threshold reached from lambda"); };

// так не подойдет, не сможем отписаться

// ....
counter.ThresholdReached -= counterOnThresholdReached;
```

Самостоятельно:

- реализовать интерфейс `INotifyPropertyChanged` на произвольном классе, продемонстрировать его работу
- реализовать очередь которая генерирует событие когда кол-во объектов в ней превышает `n` и событие когда становится пустой
- реализовать класс анализирующий поток чисел и если число отличается более чем `x` процентов выбрасывает событие

[Полный пример](#)

List, Dictionary

Класс List<T>, типизированный список объектов.

Основные операции: вставка, удаление, сортировка, получить индекс элемента, поиск.

Чаще всего употребляется когда изначально неизвестно число объектов для хранения.

Пример использования:

```
static void Main(string[] args)
{
    List<int> numbers = new List<int>() { 1, 2, 3, 4 };

    numbers.Add(5); // добавление элемента

    //добавляем несколько элементов
    numbers.AddRange(new int[] { 6, 7, 8 });

    // вставляем на первое место в списке число 666
    numbers.Insert(0, 666);

    numbers.RemoveAt(1); // удаляем второй элемент

    numbers.Sort(); // сортируем элементы

    foreach (int i in numbers)
    {
        Console.WriteLine(i);
    }
}
```

Класс Dictionary<TKey, TValue>, типизированный словарь.

Словарь хранит в себе набор элементов, где каждый элемент это ключ и значение.

Основные операции: вставка и удаление элемента по ключу.

Ключи в данной коллекции уникальны, т.е. нельзя два и более раз вставить один и тот же ключ. Для определения уникальности используются методы GetHashCode и Equals, это означает, что если в качестве ключей вы собираетесь использовать свои классы, то вам необходимо переопределить данные методы.


```

static void Main(string[] args)
{
    Dictionary<string, string> phoneBook = new Dictionary<string, string>
    {
        {"77712345", "Иванов И.И"},
        {"77812345", "Петров П.П"},
        {"77912345", "Сидоров С.С"}
    };

    phoneBook.Add("77512345", "Васильев В.В");
    phoneBook["77412345"] = "Алексеев А.А";
    // если ключ "77412345" уже есть в словаре, то произойдет обновление,
    // если нет - вставка.

    phoneBook.Remove("77712345");

    foreach(var pair in phoneBook)
    {
        Console.WriteLine("{0} - {1}", pair.Key, pair.Value);
    }
}

```

Практическая часть:

В проекте «Services» реализовать сервис «TestDataGenerator», предоставляющий методы:

- а) генерации коллекции 1000 клиентов банка;
- б) генерации словаря в качестве ключа которого применяется номер телефона клиента, в качестве значения сам клиент;
- в) генерации коллекции 1000 сотрудников банка.

Готовый результат работы сервиса протестировать в методе «Main», осуществить:

- а) пользуясь инструментом “Stopwatch”, провести замер времени выполнения поиска клиента по его номеру телефона среди элементов коллекции;
- б) провести замер времени выполнения поиска клиента по его номеру телефона, среди элементов словаря;
- в) выборку клиентов, возраст которых ниже определенного значения;
- б) поиск сотрудника с минимальной заработной платой;

Вопросы для самоконтроля:

- Чем отличается список от массива?
- В чем ключевое преимущество словаря перед списком?
- Что можно использовать в качестве ключа словаря?
- Какие важные методы должны быть реализованы для корректной работы словаря?

Самостоятельно:

Изучить работу с генератором фейковых данных «Bogus». Применяя функционал, предоставляемый инструментом «Bogus» усовершенствовать методы сервиса, для генерации коллекций сущностей, содержащих случайные значения.

Ресурсы:

- [Официальная документация по List](#)
- [Официальная документация по Dictionary](#)
- [C# и .NET | LINQ \(metanit.com\)](#)

Эквивалентность, Equals, GetHashCode.

Как мы знаем, в языке C# все типы являются наследниками базового класса `object`. В нём есть три виртуальных метода: `ToString()`, `Equals()`, `GetHashCode()`. Рассмотрим методы `Equals()` и `GetHashCode()` для ссылочных типов и типов значений.

1. Эквивалентность - оператор “==”

Для определения эквивалентности (равенства) любых объектов в языке C# можно использовать несколько разных способов, но в рамках курса мы рассмотрим следующие 2 способа:

- использование оператора “==” между сравниваемыми объектами;
- использование метода `Equals()` на одном из сравниваемых объектов, передав в качестве параметра второй объект.

По умолчанию оба этих способа вернут одинаковый результат. Для примера сравним обоими способами 2 целочисленных значения (переменные значимого типа) и 2 экземпляра класса `MyClass()` (переменные ссылочного типа), инициализированных одинаковыми значениями свойств. Порядок указания переменных для этих методов значения не имеет.

```
var int1 = 1;
var int2 = 1;
Console.WriteLine(int1 == int2);           //true
Console.WriteLine(int1.Equals(int2));      //true

var ref1 = new MyClass()
{
    Id = 1,
    Prop1 = "prop1"
};
var ref2 = new MyClass()
{
    Id = 1,
    Prop1 = "prop1"
};
Console.WriteLine(ref1 == ref2);           //false
Console.WriteLine(ref1.Equals(ref2));      //false
```

Наблюдаемый на примере выше, результат объясняется тем что:

- для значимых типов эквивалентность определяется путем сравнения значений,
- для ссылочных типов эквивалентность определяется путем сравнения ссылок сравниваемых объектов.

Оператор “==” доступен для использования для всех значимых типов, кроме структур. Если есть необходимость определять эквивалентность для структур, вам нужно написать перегрузку оператора “==” или использовать метод **Equals()**.

Пример перегрузки оператора “==” представлен в листинге ниже. Обязательное условие реализации перегрузки оператора “==” - описать перегрузку оператора “!=”. Это необходимо для обеспечения логической целостности - если равенство объектов определяется некоторым специфическим образом, то таким же образом должно определяться их различие.

```
public class BaseClass
{
}

public class MyClass : BaseClass
{
    public int Id { get; set; }
    public string Str1 { get; set; }

    public static bool operator == (MyClass first, MyClass second)
    {
        var idEquals = first.Id.Equals(second.Id);
        var prop1Equals = first.Prop1.Equals(second.Prop1);
        return idEquals && prop1Equals;
    }

    public static bool operator != (MyClass first, MyClass second)
    {
        var idEquals = first.Id.Equals(second.Id);
        var prop1Equals = first.Prop1.Equals(second.Prop1);
        return idEquals && prop1Equals;
    }
}
```

Теперь, когда мы предопределили оператор “==” подставив ему наш собственный алгоритм сравнения, результат его применения на экземплярах класса “MyClass” будет выглядеть следующим образом:

```

var ref1 = new MyClass() { Id = 1, Prop1 = "prop1" };
var ref2 = new MyClass() { Id = 1, Prop1 = "prop1" };

//сравнение MyClass
Console.WriteLine(ref1 == ref2); //true

//приведение каждого операнда к BaseClass перед сравнением
Console.WriteLine(ref1 as BaseClass == ref2 as BaseClass); //false

```

2. Метод Equals

Метод **Equals()**, виртуальный, и его базовая реализация это просто проверка равенства ссылок оператором “==”. Но если мы создаём свой собственный класс и нам необходимо реализовать возможность проверки идентичности объектов, нам нужно переопределить в своем классе метод **Equals()**, который принимает 1 аргумент, объект с которым будет сравниваться текущий объект.

Приведем пример переопределения данного метода, на классе описывающего простую геометрическую фигуру:

```

public class Figure
{
    public int Length { get; }
    public int Width { get; }

    public Figure(int length, int width)
    {
        Length = length;
        Width = width;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (!(obj is Figure))
            return false;

        var figure = (Figure) obj;
        return figure.Length == Length && figure.Width == Width;
    }
}

```

После переопределения, сравнение посредством метода **Equals()** дает корректный результат подчиняясь описанному нами алгоритму сравнения. В отличии от оператора “==” который в данном случае остался не переопределенным.

```
class Program
{
    static void Main(string[] args)
    {
        Figure f1 = new Figure (10, 20);
        Figure f2 = new Figure (10, 20);
        Figure f3 = new Figure (20, 30);
        Figure f4 = f1;
        var b1 = f1.Equals(f2); // true
        var b2 = f1.Equals(f3); // false
        var b3 = f1 == f2; // false
        var b4 = f1 == f4; // true
    }
}
```

3. Метод GetHashCode

Данный метод возвращает некоторое число для заданного объекта. Корректная реализация данного метода должна обладать двумя свойствами:

- Метод должен возвращать одинаковое число при вызове этого метода на одном и том же объекте;
- Метод должен возвращать одинаковое число при вызове этого метода на эквивалентных (равных по полям, свойствам) объектах.

Базовая реализация данного метода не удовлетворяет второму условию, поэтому для своих классов, необходимо переопределять этот метод.

Переопределим метод GetHashCode для ранее созданного класса Figure:

```
public override int GetHashCode()
{
    return Length + Width;
}
```

И протестируем работу переопределенного метода на нескольких экземплярах этого класса:

```

class Program
{
    static void Main(string[] args)
    {
        Figure f1 = new Figure (10, 20);
        Figure f2 = new Figure (10, 20);
        Figure f3 = new Figure (20, 30);

        var hash1 = f1.GetHashCode(); // 30
        var hash2 = f2.GetHashCode(); // 30
        var hash3 = f3.GetHashCode(); // 50
    }
}

```

Как видно из примера, теперь метод соответствует предъявленным выше требованиям. Корректная работа методов **Equals()** и **GetHashCode()** очень важна, так как эти методы неявно применяются при работе с коллекциями, например при сортировке списка, или поиске по словарю и хэш таблице.

4. Equals и GetHashCode применительно к структурам

Метод **Equals()** для структур по умолчанию сравнивает значения всех полей экземпляров структур. Каким образом сравниваются значения полей? У встроенных типов значений .net framework, таких как: int, char, double и так далее, есть корректная, переопределенная реализация сравнения. Однако если полем структуры является ссылочный тип, то как мы видели выше - базовая реализация **Equals()** сравнивает ссылки.

Продemonстрируем это на примере, сделаем нашу геометрическую фигуру теперь структурой:

```

struct Figure
{
    public int Length { get; }
    public int Width { get; }

    public Figure(int length, int width)
    {
        Length = length;
        Width = width;
    }
}

```

И проверим работу метода **Equals()**:

```
Figure f1 = new Figure(1, 2);
Figure f2 = new Figure(1, 2);
Figure f3 = new Figure(3, 4);

f1.Equals(f2) // true
f1.Equals(f3) // false
```

Пока все корректно. Теперь добавим в нашу структуру какой-нибудь ссылочный тип у которого не переопределен метод **Equals()** и посмотрим на результат сравнения:

```
struct Figure
{
    public int Length { get; }
    public int Width { get; }
    public Color color { get; set; }

    public Figure(int length, int width, Color color)
    {
        Length = length;
        Width = width;
        Color = color;
    }
}

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

```
Color green= new Color
{
    Value = "Зеленый"
};

Color black= new Color
{
    Value = "Зеленый"
};

Figure f1 = new Figure(1, 2, green);
Figure f2 = new Figure(1, 2, black);
Figure f3 = new Figure(3, 4, green);

f1.Equals(f2) // false
f1.Equals(f3) // false
```


Как мы видим результат сравнения false. Это не то, что нам нужно, ведь названия фигур одинаковые. Переопределим метод Equals в классе Name и посмотрим на результат сравнения снова.

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

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (!(obj is Name))
            return false;

        var name = (Name) obj;
        return name.Value == Value;
    }
}
```

```
f1.Equals(f2) // true
f1.Equals(f3) // false
```

Теперь мы видим, что всё работает корректно.

Метод GetHashCode хоть и переопределен для типов значений, но он так же не удовлетворяет условиям перечисленным для ссылочного типа и для собственных типов значений его также нужно переопределять.

Практическая часть:

- добавить новую модель - класс “Account” (банковский счет клиента), со свойствами Currency, Amount;
- в классе “TestDataGenerator” реализовать метод генерирующий словарь, где в качестве ключа находятся клиенты, в качестве значения их банковский счет;
- попробовать получить счет клиента применив клиента в качестве ключа;
- переопределить методы Equals и GetHashCode, и повторить попытку
- усложнить задачу на случай когда у клиента несколько банковских счетов;
- реализовать метод добавления новых счетов текущему клиенту.

- **Вопросы для самоконтроля:**

- Возникнут ли проблемы, если перед сравнением мы приведем объекты к родительскому типу?
- Всегда ли необходимо переопределять методы **Equals()** и **GetHashCode()**?
- Как происходит сравнение экземпляров пользовательского класса, если в нем не переопределен оператор ?
- Можно ли в словарь добавить несколько записей с одинаковым ключом?
- Что необходимо сделать, чтобы вызвался переопределенный метод Equals при сравнении через оператор “==”.

Самостоятельно:

- Выяснить как работает оператор “==” у типа string. Объяснить результат работы;

Ресурсы:

- [Метод GetHashCode](#)
- [Метод Equals](#)

Comparable - сравнение объектов

Например, когда вы вызовете метод `OrderBy` на коллекции объектов то, чтобы понять какой объект больше ($>$) или меньше ($<$), вызовется метод **`int CompareTo(object)`**, и в зависимости от результата произойдет сортировка.

- <0 меньше
- $==0$ равны
- >0 больше

Для всех встроенных типов разумеется этот интерфейс определен и поэтому коллекции стандартных типов можно спокойно сортировать.

Самостоятельно:

- Реализуйте класс произвольной фигуры (треугольник, квадрат, круг), определите **`CompareTo<T>`**, сравнение производим по площади фигуры, затем генерируйте 10 объектов и отсортируйте в порядке убывания.
- **`IComparer`** изучить самостоятельно для чего используется, реализовать пример
- Можно ли сортировать последовательность без реализованного **`CompareTo`** ?

[Полный пример](#)

Обработка исключений (Exception handling)

Исключение — это любое состояние ошибки или непредвиденное поведение, возникающее при выполнении программы.

Исключения могут возникать из-за сбоя в вашем или вызываемом коде (например, в общей библиотеке), недоступности ресурсов ОС, неожиданных состояний, возникающих в среде выполнения и по другим причинам. После некоторых из этих состояний приложение может восстановиться, после других — нет. Как правило, вы можете перехватить большинство исключений и восстановить работу приложения, но не после системных исключений (например, [System.StackOverflowException](#)).

В .NET исключение — это объект, наследуемый от класса [System.Exception](#). Исключение создается из области кода, где произошла проблема. Исключение передается вверх по стеку до тех пор, пока его не обработает приложение либо программа не завершится.

Пример кода, генерирующего исключение:

```
class Program
{
    static void Main(string[] args)
    {
        int x = 5;
        int y = x / 0;
        Console.WriteLine($"Результат: {y}");
        Console.WriteLine("Конец программы");
        Console.Read();
    }
}
```

В данном случае происходит деление числа на 0, что приведет к генерации исключения. При запуске этой программы произойдет ее аварийное завершение, так как возникшее исключение не перехватывается и не обрабатывается.

Язык C# предоставляет разработчикам возможности для перехвата и обработки таких ситуаций. Для этого в C# предназначена конструкция `try...catch`.

Чтобы перехватить и обработать эту исключительную ситуацию необходимо поместить все операторы кода, которые могут вызвать исключение, в блок `try`, а операторы, которые обрабатывают исключения, поместить в одном или нескольких блоках `catch` под блоком `try`.

Пример перехвата и обработки возникшего исключения:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
        catch
        {
            Console.WriteLine("Возникло исключение");
        }

        Console.WriteLine("Конец программы");
        Console.Read();
    }
}
```

Выполнение этого кода не приведет к аварийному завершению программы.

В общем случае вместо использования базового оператора catch рекомендуется перехватывать исключения определенного типа.

При возникновении исключения оно передается вверх по стеку, и каждый блок catch получает возможность обработать его. Важен порядок операторов catch. Размещайте блоки catch, предназначенные для определенных исключений, до общего блока перехвата исключений.

Пример перехвата и обработки определенного исключения:

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
    }
}
```

```

        catch (DivideByZeroException)
        {
            Console.WriteLine("Перехвачено исключение
DivideByZeroException");
        }
        catch (Exception exception)
        {
            Console.WriteLine($"Перехвачено исключение:
{exception.Message}");
        }

        Console.WriteLine("Конец программы");
        Console.Read();
    }
}

```

В этом случае исключение перехватит блок `catch(DivideByZeroException)`, программа выведет строку "Перехвачено исключение `DivideByZeroException`".

Если бы код генерировал бы другой тип исключения, например, `System.InvalidCastException`, то исключение было бы перехвачено блоком `catch(Exception)`, так как тип `System.Exception` является базовым для всех типов исключений.

Исключение можно вызвать явным образом с помощью оператора `throw`. Перехваченное исключение можно вызвать повторно с помощью оператора `throw`. При написании кода рекомендуется добавлять сведения в исключение, которое выдается повторно, чтобы предоставить дополнительную информацию при отладке.

Явная генерация исключения:

```

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 0;
            if (x == 0)
            {
                throw new InvalidOperationException("Недопустимое
значение переменной");
            }
        }
    }
}

```

```

        catch (InvalidOperationException exception)
        {
            Console.WriteLine($"Перехвачено исключение:
                               {exception}");

            throw;
        }

        Console.WriteLine("Конец программы");
        Console.Read();
    }
}

```

В этом коде мы явно генерируем исключение `InvalidOperationException`. Это исключение будет перехвачено в блоке `catch (InvalidOperationException exception)`, но в этом же блоке будет повторно сгенерировано с помощью оператора `throw`. Такой подход применяется, чтобы обработать возникшее исключение и расширить его дополнительными данными, либо выбросить другое исключение, а затем передать его на дальнейшую обработку вверх по стеку. В нашем примере это приведет к аварийному завершению программы.

При возникновении исключения выполнение останавливается, и управление передается соответствующему обработчику исключений. Часто это означает, что ожидаемые вами строки кода пропускаются. Даже при возникновении исключения требуется определенная очистка ресурсов, например закрытие файла. Для этого можно использовать блок `finally`. Следует помнить, что код в блоке `finally` будет выполнен, если возникшее исключение в итоге будет перехвачено здесь же (в блоке `catch` перед блоком `finally`) или вверх по стеку.

Пример использования блока `finally`:

```

class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 0;
            if (x == 0)
            {
                throw new InvalidOperationException("Недопустимое значение переменной");
            }
        }
    }
}

```

```

    }
    catch (InvalidOperationException exception)
    {
        Console.WriteLine($"Перехвачено исключение: {exception}");
    }
    finally
    {
        Console.WriteLine("finally блок выполняется всегда");
    }

    Console.WriteLine("Конец программы");
    Console.Read();
}
}

```

Практическая часть:

Добавить новые сервисы для работы с клиентами и сотрудниками банка (ClientService, EmployeeService), с учетом что после будет добавлено хранилище.

В рамках ClientService реализовать метод добавления новых сотрудников. В методе предусмотреть валидацию:

- выбрасываем исключение если клиент моложе 18 лет (добавить собственный класс исключения);
- выбрасываем исключение если у клиента нет паспортных данных;
- реализовать механизм по аналогии для сотрудников.
- самостоятельно реализовать аналогичный подход для работы с сотрудниками банка.

Вопросы для самоконтроля:

- Почему “исключение” назвали исключением?
- Что подразумевается под обработкой исключения?
- Можем ли мы добавить в программу собственный тип исключения?

Самостоятельно:

- Класс [System.Exception](#)
- Механика обработки исключений (try, catch, finally, обработка определенных исключений)

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/api/system.exception>
- <https://docs.microsoft.com/ru-ru/dotnet/standard/exceptions/>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/exceptions/>
- <https://docs.microsoft.com/ru-ru/dotnet/standard/exceptions/how-to-create-user-defined-exceptions>
- <https://metanit.com/sharp/tutorial/2.14.php>

Базовые интерфейсы для работы со списками (IEnumerable, IEnumerator)

Основой для реализации большинства коллекций является интерфейс **IEnumerable**. Благодаря тому что коллекция реализует этот интерфейс, появляется возможность перебирать её элементы. **IEnumerable** выглядит следующим образом:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

То есть интерфейс обязывает коллекцию реализовать метод **GetEnumerator**. Этот метод в свою очередь должен вернуть объект реализацию **IEnumerator**, то есть объект – перечислитель. Данный объект, который предоставляет методы для последовательного перебора и получения всех элементов коллекции. Сам интерфейс выглядит следующим образом:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current {get;}
    void Reset();
}
```

Метод **MoveNext** предназначен для перемещения указателя на следующую позицию и возвращения **true**, если операция успешно завершена и **false**, если коллекция завершена.

Свойство **Current** возвращает тот элемент коллекции, на который в данный момент указывает указатель.

Метод **Reset** возвращает указатель в дефолтное состояние.

Конечно, в своей реализации **IEnumerator** вы можете заложить другое поведение коллекции, но учитывайте, что эти методы неявно применяются при переборе коллекции в конструкции **foreach**, а также всеми методами

расширения, которые перебирают элементы коллекции (такие как **where**, **any**, **orderBy** и др.).

Пример применения реализации **IEnumerable**:

```
public class People : IEnumerable<Person>{...}

Static void Main()
{
    Person[] peopleArray = new Person[3]
    {
        new Person("John", "Smith"),
        new Person("Jim", "Johnson"),
        new Person("Sue", "Rabon")
    };

    People peopleList = new People(peopleArray);

    foreach (Person p in peopleList)
    {
        Console.WriteLine(p.FirstName + " " + p.LastName);
    }
}

/* This code produces output similar to the following:
   John Smith
   Jim Johnson
   Sue Rabon */
```

В примере выше видно как мы перебираем все члены множества **People** с помощью оператора **foreach**. Оператор **foreach** "за ширмой" вызывает метод **GetEnumerator()** и, получив его, выполняет полный обход множества. **Стоит запомнить, что пока существует активный enumerator, изменять коллекцию (источник данных) нельзя! .NET выбросит исключение!**

Также отдельно стоит упомянуть что о том, что ряд операций над **IEnumerable** "ленивые". Это значит, что выполнение запроса происходит только во время перебора коллекции. Условно LINQ запрос делится на 3 части:

1. Получение источника данных
2. Создание запроса
3. Выполнение запроса и получения результатов

Разберем на практике:

```
Public class People : IEnumerable<Person>{...}
    Static void Main()
    {
        // Создание источника данных
        Person[] peopleArray = new Person[3]
        {
            new Person("John", "Smith"),
            new Person("Jim", "Johnson"),
            new Person("Sue", "Rabon")
        };

        // Создание запроса
        filteredPeople = peopleArray.Where(u => u.FirstName == "John");

        // Выполнение запроса
        foreach (Person p in filteredPeople)
        {
            Console.WriteLine(p.FirstName + " " + p.LastName);
        }
    }
}
```

Практическая часть:

В проект “Services” добавить класс “ClientStorage” представляющий хранилище клиентов.

- в хранилище добавить readonly список для хранения клиентов банка, реализуем публичные методы для управления списком;
- экземпляр готового хранилища передаем через параметры конструктора в сервис работы с клиентами;
- в сервисе реализуем метод получения выборки из хранилища с применением фильтра:
- в классе тестов, проверяем работу системы, добавляем клиентов в хранилище и делаем выборки из нее;
- найдите самого молодого клиента в коллекции;
- найдите самого старого клиента в коллекции;
- вычислите средний возраст клиентов в коллекции;
- самостоятельно повторить работу на примере сотрудников банка.

Вопросы для самоконтроля:

- 1) Назовите уже существующие в **.Net** классы – реализации интерфейса **IEnumerable**. Как убедиться, что они действительно его реализуют?
- 2) Как вы понимаете ленивое исполнение запросов?
- 3) Когда происходит перечисление коллекции?

Ресурсы:

- [С# и .NET | Обработка исключений \(metanit.com\)](#)
- [С# и .NET | Типы исключений. Класс Exception \(metanit.com\)](#)
- [С# и .NET | Создание классов исключений \(metanit.com\)](#)
- [С# и .NET | Отложенное и немедленное выполнение LINQ \(metanit.com\)](#)
- [LINQ explained with sketches \(steven-giesel.com\)](#)

Generic Type, Generic Member

Предположим, что у нас есть некий метод, который по задаче должен принимать один аргумент нескольких разных типов. Одним из решений будет определить этот аргумент как тип `Object`. После чего уже в теле метода типизировать этот `object` в нужный нам тип.

```
class Test
{
    static void Main()
    {
        Show(4);
        Show("четыре");
    }

    static void Show(object id)
    {
        var value = id;
        var count = 5 + (int)id;
        Console.WriteLine(count);
    }
}
```

Таким образом мы прибегнем к таким процессам как упаковка и распаковка. Как уже говорилось ранее - данные процессы ресурсозатратным и небезопасны.

Для избежания этих проблем используется `Generic Type`, или же обобщенный тип.

Угловые скобки в описании метода указывают, что метод является обобщенным, а тип `T` в угловых скобках условно обозначает тип который в дальнейшем будет использоваться данным методом. Если требуется несколько типов, они указываются через запятую в угловых скобках, различными заглавными буквами.

Как видно из примера, при вызове метода и передаче ему аргумента конкретного типа, сигнатура метода конкретизируется:

```
class Test
{
    static void Main()
    {
        Show(4);           // 4, type: System.Int32
        Show("четыре");    // четыре, type: System.String
    }
}
```

```

static void Show<T>(T id)
{
    var value = id;

    var type = typeof(T);

    Console.WriteLine(id + ", type: " + type);
}
}

```

При работе с обобщенными классами, тип аналогичным образом указывается в угловых скобках рядом с названием класса, и применяется в работе внутри класса:

```

class Person<T>
{
    public T Id { get; set; }
    public string Name { get; set; }
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}

```

Generic Type можно ограничивать, “закрывать” определенными типами. Ограничение позволяет на этапе написания кода указать, какие типы данных могут быть применены при вызове метода.

Рассмотрим на примере, опишем несколько простых классов, некоторые из которых будут реализовывать интерфейс:

```

public class Cat : IWalk
{
    public void Walk()
    {
        Console.WriteLine("Cat walking");
    }
}

```

```
public class Dog : IWalk
{
    public void Walk()
    {
        Console.WriteLine("Dog walking");
    }
}
```

```
public class Fish
{
}
```

```
public interface IWalk
{
    void Walk();
}
```

Далее опишем обобщенный метод и ограничим его при помощи ключевого слова “where”.

Сигнатура ограничений выглядит следующим образом: после закрывающейся круглой скобки пишется оператор `where`, имя обобщенного типа, двоеточие, затем через запятую перечисляются типы которые допускаются для работы с данным обобщением. Кроме конкретных классов могут применяться конкретные интерфейсы, а также универсальные параметры:

- Классы
- Интерфейсы
- `class` - универсальный параметр должен представлять класс
- `struct` - универсальный параметр должен представлять структуру
- `new()` - универсальный параметр должен представлять тип, который имеет общедоступный (`public`) конструктор без параметров

Если ограничений несколько - их указывают через запятую.

```
static void WhoIsGoing<T>(T animal) where T : IWalk
{
    animal.Walk();
}
```


После чего попробуем передать в этот метод экземпляры созданных ранее классов:

```
class Test
{
    static void Main()
    {
        var cat = new Cat();
        var dog = new Cat();
        var fish = new Fish();

        WhoIsGoing(cat);
        WhoIsGoing(dog);
    }

    static void WhoIsGoing<T>(T animal) where T : IWalk
    {
        animal.Walk();
    }
}
```

Метод **WhoIsGoing()** без проблем принимает в качестве параметров классы Cat и Dog, так как они реализуют интерфейс IWalk, однако при попытке передать экземпляр класса Fish возникнет ошибка, так как данный класс не подходит условиям ограничения.

Теперь рассмотрим пример использования нескольких обобщенных типов:

```
public class Test
{
    static void Main()
    {
        CalculateDistance(
            new MovementSpeed() {Speed = 10},
            new Time() {SpentTime = 2}
        );    //Distance: 20
    }

    static void CalculateDistance<T, U>(T speed, U time)
    where T : MovementSpeed where U : Time
    {
        var distance = speed.Speed * time.SpentTime;
        Console.WriteLine("Distance: " + distance);
    }
}
```

```
public class MovementSpeed
{
    public double Speed { get; set; }
}
```

```
public class Time
{
    public double SpentTime { get; set; }
}
```

Если класс использует несколько универсальных параметров, то последовательно можно задать ограничения к каждому из них, в чем мы убедились на примере.

Практическая часть:

В сервисе “BankService” реализуем методы:

- а) AddBonus принимает на вход наследника класса Person и добавляет ему бонус;
- б) AddToBlackList<T>(T person) - добавляет сущность в список хранящийся в этом же классе;
- в) IsPersonInBlackList<T>(T person) - проверяет присутствует ли сущность в черном списке.

Вопросы для самоконтроля:

- В чем преимущество обобщенных типов над упаковкой и распаковкой?
- Можно ли передать в класс в обобщенный метод, если в качестве ограничения стоит интерфейс?
- В теме говорится про закрытый generic, а что такое открытый generic?

Ресурсы:

- [C# и .NET | Обобщения \(metanit.com\)](https://metanit.com/ru/learn/csharp/generics/11/)
- [C# и .NET | Ограничения обобщений \(metanit.com\)](https://metanit.com/ru/learn/csharp/generics/12/)
- [C# и .NET | Наследование обобщенных типов \(metanit.com\)](https://metanit.com/ru/learn/csharp/generics/13/)

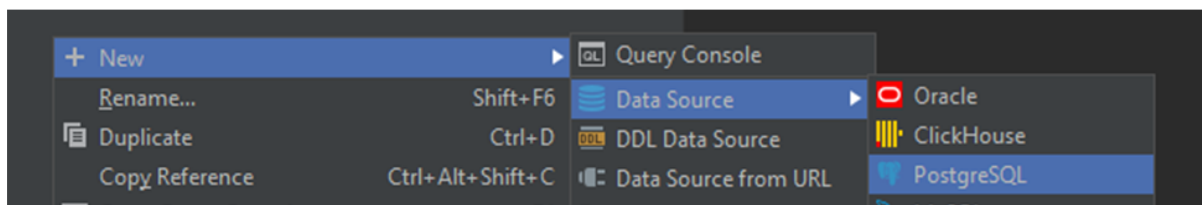
Работа с базой данных. Проектирование, нормализация, DataGrip

База данных представляет собой хранилище на жестком диске, в котором данные хранятся в виде структурированных по определенным правилам таблиц.

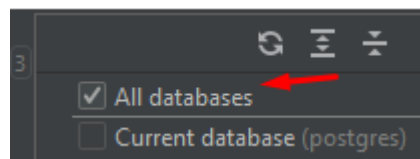
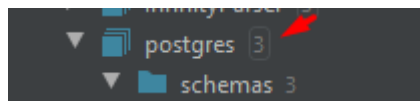
В отличие например от Excel, применение базы данных (в нашем случае Postgres) предоставляет набор инструментов, увеличивающих безопасность и эффективность работы с данными. Однако даже для рядового ознакомления с этими нюансами необходим отдельный курс лекций. В рамках наших лекций мы проведем только первичное (поверхностное) знакомство с упомянутыми системами а в качестве инструмента для работы с базой будем применять DataGrip. Соответственно, к данному моменту, Postgres и DataGrip должны быть установлены и настроены.

1. Создание новой базы

После первого запуска, DataGrip не визуализирует ни одной базы. Для создания новой базы нажмем правой кнопкой на панели обозревателя в DataGrip, в выпадающем списке выберем пункт new => Data Source => PostgreSQL.



После чего откроется меню для введения имени будущей базы и настроек доступа к ней. Заполним поля: Name, User, Password, Database, прочее оставляем по умолчанию и нажмем кнопку ОК.

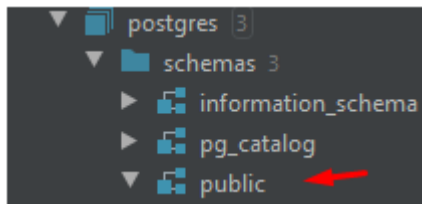


В панели слева появится новый каталог с именем который мы присвоили нашему источнику данных. Нажмем на значок фильтра рядом с названием источника и выберем “All databases”.

На данный момент источник данных уже содержит базу, с именем которое мы указали в поле Database когда заполняли форму.

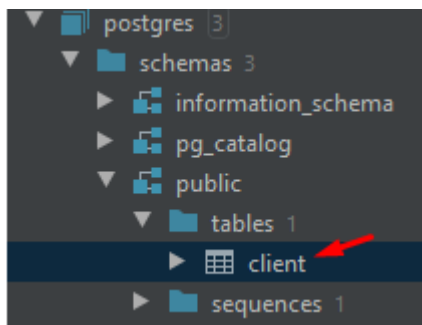
При желании мы можем добавить еще одну, правой кнопкой на имени источника new => DataBase.

2. Создание новой таблицы



Для начала работы с таблицами, найдем в нашей базе папку «public».

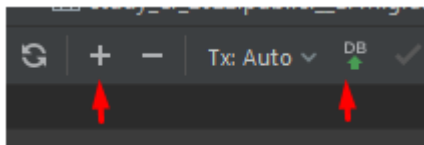
На данный момент она пуста. Нажмем правой клавишей на название этой папки и выберем пункт New => Table. При этом появляется окно, в котором можем произвести подробную настройку создаваемой таблицы в соответствии с архитектурой, сущности которую мы намерены в ней хранить.



Создадим таблицу “client”, добавим столбец “id” установим галочки: *в качестве ключа, обязательно, не null, автоинкремент*. Кроме того добавим столбцы имя и возраст. Имена таблиц и столбцов принято давать в нижнем регистре, без заглавной буквы. Обратите внимание в нижнем окне происходит трансляция наших действий в команды SQL. Будьте осторожны, в SQL свои типы данных!

После нажатия кнопки “Execute” в папку «public» добавиться сконструированная нами таблица.

После добавления мы можем найти нашу таблицу в папке «tables», и увидеть ее графическую структуру нажав на нее два раза левой клавишей.



Добавим в таблицу несколько записей вручную. Для этого достаточно нажать на плюс на панели инструментов, в результате в таблицу добавляется пустая строка, которую мы можем заполнить выделив соответствующую ячейку. После того как все ячейки заполнены, необходимо нажать зеленую стрелочку с надписью “DB”, это зафиксирует внесенные в таблицу изменения.





3. Нормализация

Данные в базе хранятся определенным образом. Особенности хранения данных в таблице представляют собой ряд правил, призванных облегчить, обезопасить, повысить продуктивность при работе с данными.

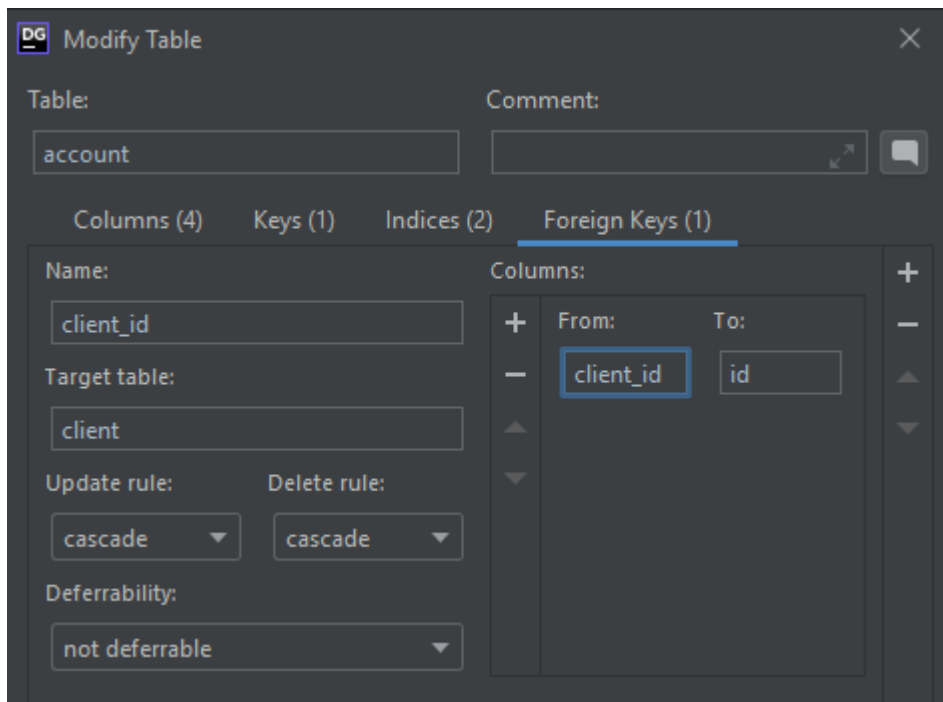
Перечислим некоторые из них:

1. Как вы могли заметить в рамках предыдущей темы, наша таблица состоит из столбцов каждый из которых отвечает свойству в модели данных (классу для которой создается таблица - Employee, Client).
2. Добавление новой записи в таблицу подразумевает добавление новой строки.
3. Таблица содержит по крайней мере один столбец хранящий уникальные значения для однозначной идентификации записи (строки), этот идентификатор называется “первичным ключом”. Это может быть email или номер паспорта сотрудника (клиента), но чаще применяется специальный тип данных Guid. Подробнее про Guid смотри тут: [Евангелие от GUID / Хабр \(habr.com\)](#).
4. Если нам необходимо связать записи одной таблицы с записями другой, например, связать клиента с его банковским счетом – в таблице, хранящей данные о счете «account», кроме столбца с первичными ключами, добавляется столбец хранящий идентификатор записи в таблице «client» - его называют “Внешним ключом” .

Q- <Filter Criteria>

	 id	 name	 amount	 client_id
1	2	USD	42	8

Добавить ключ можно воспользовавшись вкладкой “Foreign Keys” в окне создания/редактирования структуры таблицы. Для этого мы должны указать с какой таблицей нас связывает создаваемый ключ (поле “Target table”), сценарий поведения при изменении или удалении записи с которой связываемся (“Update rule” и “Delete rule” соответственно), какие столбцы связываем (“From” - название столбца в текущей таблице, “To” - в таблице с которой связываемся):



Теперь, если наш клиент заводит второй лицевой счет, мы добавляем в таблицу «account» новую строку у которой в столбце “client_id” будет тот же идентификатор, указывающий на одного и того же клиента.

Q* <Filter Criteria>

	id	name	amount	client_id
1	2	USD	42	8
2	3	MDL	223	8

Неправильная альтернатива – хранить значение идентификатора лицевого счета в одной из ячеек таблицы «Client» в виде строки (string) и перечислять новые через запятую.

Стоит упомянуть, что кроме рассмотренного примера, таблицы также могут быть связаны ассоциативной связью один-к-одному и многие-ко-многим. Но для простоты изложения материала мы не будем рассматривать эти примеры. Подробнее с этими тонкостями можно ознакомиться тут : [Связи между таблицами базы данных / Хабр \(habr.com\)](https://habr.com/ru/articles/2017/05/12/101111/)

Практическая часть:

а) создать новую базу данных для проекта “Банковская система”, добавить в нее таблицы в соответствии с сущностями (Employee , Client, Account);

б) связать таблицы клиент и банковский счет внешним ключем (у одного клиента много счетов);

г) заполнить таблицы записями, убедиться что без внешнего ключа (существующий Id клиента) запись в таблице “account” не удастся добавить.

Вопросы для самоконтроля:

- Для чего нужна БД?
- В каком стиле принято именовать таблицы и их столбцы?
- Что такое primary key? Для чего нужен?
- Что такое foreign key? Для чего нужен?

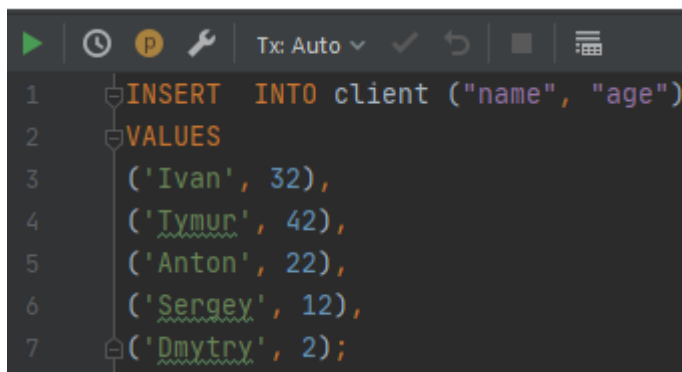
SQL практика

На прошлой лекции мы заполняли таблицу данными вручную. Однако, этот подход применим только для проведения небольших тестов. В рамках текущей темы мы познакомимся с основными SQL (Structured Query Language — «язык структурированных запросов») запросами, позволяющие автоматизировать работу с базой данных.

Для начала работы с SQL в Data Grip, воспользуемся консолью (правой клавишей на таблице, New => Query Console).

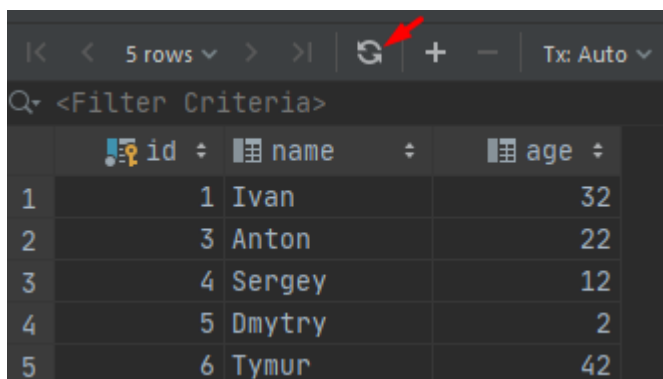
В открывшейся вкладке нам доступна работа с функционалом базы посредством языка SQL.

Пример – заполним таблицу записями, выполнив следующий запрос:



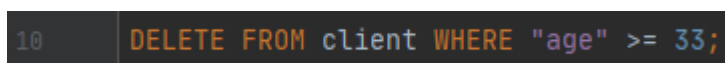
```
1 INSERT INTO client ("name", "age")
2 VALUES
3 ('Ivan', 32),
4 ('Tymur', 42),
5 ('Anton', 22),
6 ('Sergey', 12),
7 ('Dmytry', 2);
```

Не забывайте нажимать кнопку “обновить” над таблицей, чтобы видеть актуальные данные:



<Filter Criteria>			
	id	name	age
1	1	Ivan	32
2	3	Anton	22
3	4	Sergey	12
4	5	Dmytry	2
5	6	Tymur	42

Пример – удалим клиента возраст которого равен или более 33 лет:



```
10 DELETE FROM client WHERE "age" >= 33;
```


Q: <Filter Criteria>			
	id	name	age
1	1	Ivan	32
2	3	Anton	22
3	4	Sergey	12
4	5	Dmytry	2

Получим список пользователей младше 30:

```
12 SELECT * FROM client WHERE "age" < 30;
```

В нижнем окне получим следующую выборку:

Output Result 11			
	id	name	age
1	3	Anton	22
2	4	Sergey	12
3	5	Dmytry	2

Подробнее о синтаксисе запросов можно почитать главу “Операции с данными” здесь: [PostgreSQL | Добавление данных. Команда Insert \(metanit.com\)](https://metanit.com/postgresql/04/)

Практическая часть:

- а) наполнить базу тестовыми данными пользуясь оператором Insert;
- б) провести выборки клиентов, у которых сумма на счету ниже определенного значения, сгруппированных в порядке возрастания суммы;
- в) провести поиск клиента с минимальной суммой на счете;
- г) провести подсчет суммы денег у всех клиентов банка;
- д) с помощью оператора Join, получить выборку банковских счетов и их владельцев.

Вопросы для самоконтроля:

- Какой оператор позволяет добавить запись в базу?
- Есть ли отличия между обозначением оператора сравнения в C# и в SQL?
- Какую операцию позволяет выполнить оператор Join?

Введение в EntityFramework, миграции

На предыдущих занятиях мы практиковались в работе с базой данных непосредственно в DataGrip, так сказать в ручном режиме. Однако, существуют инструменты позволяющие автоматизировать процесс создания, настройки, и изменения базы данных. Что существенно упрощает и ускоряет работу. Примером такого инструмента является Entity Framework с которым мы сегодня познакомимся.

EF является ORM-инструментом (object-relational mapping - объектно-реляционное отображение). Данный инструмент позволяет абстрагироваться от самой базы данных и ее таблиц, работать с данными независимо от типа хранилища (поддерживает множество различных систем баз данных). Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, то на уровне .net приложения мы уже работаем с классами с#. Таким образом если мы решим сменить целевую СУБД (систему управления базой данных), то основные изменения в проекте будут касаться прежде всего конфигурации и настройки подключения к соответствующим провайдерам. А код, который непосредственно работает с данными, получает данные, добавляет их в БД и т.д., останется прежним.

Центральной концепцией Entity Framework является понятие сущности или entity. Сущность определяет набор данных, которые связаны с определенным объектом (ФИО, номер паспорта, возраст). Поэтому данная технология предполагает работу не с таблицами, а с объектами и их коллекциями (как мы привыкли работать в рамках .Net).

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает человека, то мы можем выделить такие свойства, как имя, фамилия, рост, возраст.

И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют ключами.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-к-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

Отличительной чертой Entity Framework, как технологии ORM, является использование запросов LINQ для выборки данных из БД. С помощью LINQ мы можем создавать различные запросы на выборку объектов, в том числе связанных различными ассоциативными связями. А Entity Framework при выполнении запроса транслирует выражения LINQ в выражения, понятные для конкретной СУБД (как правило, в выражения SQL).

Для работы с базой данных PostgreSQL, создадим новый консольный проект применив и добавим в него пакеты:

- 1) **Npgsql.EntityFrameworkCore.PostgreSQL**
- 2) **Microsoft.EntityFrameworkCore.Tools**

В качестве моделей для нашей базы мы будем использовать ранее созданные сущности (Employee, Client, Account, Currency). Однако стоит отметить что Entity Framework требует определения ключа сущности для создания первичного ключа в таблице. По умолчанию при генерации бд EF в качестве первичных ключей будет рассматривать свойства с именами Id или [Имя_класса]Id (то есть EmployeeId, ClientId), в связи с чем, усовершенствуем наши модели добавив к ним новое свойство “Id” - идентификатор с типом данных GUID.

Взаимодействие с базой данных в Entity Framework происходит посредством специального класса - контекста данных. Поэтому добавим в

наш проект новый класс, который назовем ApplicationDbContext и который будет иметь следующий код:

```
using Microsoft.EntityFrameworkCore;
namespace HelloApp
{
    public class ApplicationDbContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public ApplicationDbContext()
        {
            Database.EnsureCreated();
        }
        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseNpgsql(
                "Host=localhost;
                Port=5433;
                Database=usersdb;
                Username=postgres;
                Password=пароль_от_postgres"
            );
        }
    }
}
```

Основу функциональности Entity Framework для работы с PostgreSQL составляют классы, которые располагаются в пространстве имен Microsoft.EntityFrameworkCore. Среди всего набора классов этого пространства имен следует выделить следующие:

DbContext: определяет контекст данных, используемый для взаимодействия с базой данных;

DbSet/DbSet<TEntity>: представляет набор объектов, которые хранятся в базе данных;

DbContextOptionsBuilder: устанавливает параметры подключения.

В любом приложении, работающем с БД через Entity Framework, нам нужен будет контекст (класс производный от DbContext). В данном случае таким контекстом является класс ApplicationContext.

Кроме того, для настройки подключения нам надо переопределить метод OnConfiguring класса DbContext, в который передается строка подключения. Строка подключения содержит адрес сервера (параметр Host), порт (Port), название базы данных на сервере (Database), имя пользователя в рамках сервера PostgreSQL (Username) и его пароль (Password).

В конструкторе класса контекста определен вызов метода Database.EnsureCreated(), который при создании контекста автоматически проверит наличие базы данных и, если она отсутствует, создаст ее.

EF использует ряд условностей для сопоставления классов моделей с таблицами. Например, названия столбцов должны соответствовать названиям свойств и т.д. В этом случае Entity Framework сможет сопоставить столбцы таблицы и свойства модели.

Однако предусмотрены механизмы (Fluent API, Аннотации) для добавления дополнительных правил конфигурации.

1. Fluent API

Fluent API представляет набор методов, которые определяют сопоставление между классами и их свойствами и таблицами и их столбцами. Как правило, функционал Fluent API задействуется при переопределении метода OnModelCreating класса DbContext.

Подробнее с возможностями Fluent API можно ознакомиться тут:

[Fluent API — настройка и сопоставление свойств и типов — EF6 | Microsoft Docs](#)

2. Аннотации

Аннотации представляют настройку классов моделей с помощью атрибутов. Большинство подобных атрибутов располагаются в пространстве `System.ComponentModel.DataAnnotations`, которое нам надо подключить в файл `C#` перед использованием аннотаций.

Пример:

```
using System.ComponentModel.DataAnnotations;

namespace HelloApp
{
    public class User
    {
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

В данном случае атрибут `Required` представляет аннотацию, которая указывает, что свойство `Name` обязательно должно иметь значение.

Подробнее с возможностями `Fluent API` можно ознакомиться тут:

[Code First Data Annotations - EF6 | Microsoft Docs](#)

Теперь реализуем в классе `ClientService` методы по добавлению и извлечению объектов из базы данных:

```
public class ClientService
{
    ApplicationContext _dbContext;
    public ClientService()
    {
        _dbContext = new ApplicationContext();
    }
}
```

```
public Client GetClient(Guid clientId)
{
    return _dbContext.Clients.FirstOrDefault(c => c.Id ==
clientId);
}

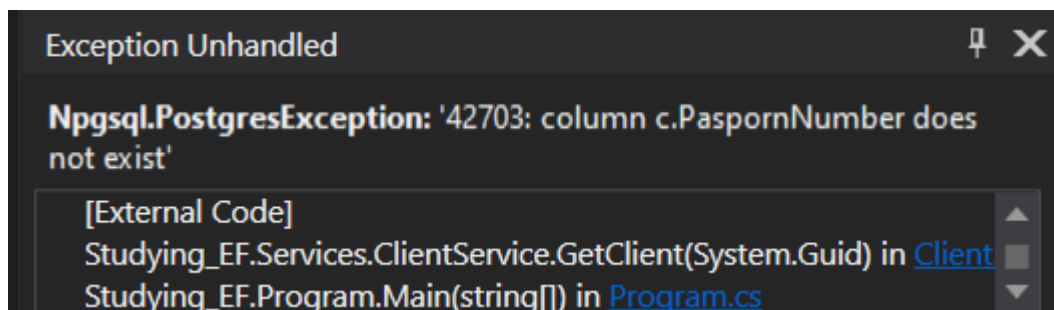
public void AddClient(Client client)
{
    _dbContext.Clients.Add(client);
    _dbContext.SaveChanges();
}
}
```

Как видно из примера благодаря посредничеству EF, мы можем взаимодействовать с базой оперируя сущностями из .Net, так словно работаем с коллекцией в памяти а не таблицей в базе.

3. Миграции

Если мы реализуем какую-то работу ведущую к изменениям в структуре базы, например: редактируем модели в Entity Framework, создаём новые, удаляем старые таблицы, то необходимо, чтобы база данных также применяла эти изменения.

Добавим нашему клиенту свойство PaspornNumber и попробуем добавить запись в базу. В попытке мы получим следующее исключение:



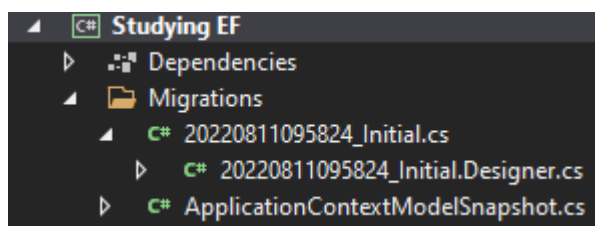
Миграция по сути представляет план перехода базы данных от старой схемы к новой. Если планируется использовать миграции, то лучше их использовать сразу при создании базы данных. Именно для этих целей мы устанавливали пакет **Microsoft.EntityFrameworkCore.Tools**.

В этом случае метод `Database.EnsureCreated()` в классе контекста нам больше не нужен. Более того при выполнении миграции этот метод вызывает ошибку.

Для создания миграции в окне Package Manager Console вводится следующая команда: *Add-Migration название_миграции*

Название миграции представляет произвольное название, главное, чтобы все миграции в проекте имели разные названия.

После этого в проект будет добавлена папка Migrations с классом миграции:



После создания миграции ее надо выполнить с помощью команды: *Update-Database*

После выполнения миграции мы найдем сгенерированную базу данных. Следует отметить, что кроме основных таблиц база данных также будет содержать дополнительную таблицу `_EFMigrationsHistory`, которая будет хранить информацию о миграциях.

Практическая часть:

В соответствии с материалом теоретической части, дополняем ранее созданные классы (Employee , Client, Account), необходимыми атрибутами, организуем связь клиента и его банковских счетов. Уделяем ранее созданную базу данных и пересоздаем ее при помощи миграции. Дорабатываем сервис для работы с клиентами банка, под работу с базой в качестве хранилища. Сервис должен реализовывать методы:

- а) получить клиента по идентификатору;
- б) добавить нового клиента (автоматически создает дефолтный лицевой счет);
- в) добавить клиенту новый лицевой счет (принимает на вход Id клиента);
- г) изменить клиента по идентификатору;
- д) удалить клиента по идентификатору.
- е) метод возвращающий список клиентов, удовлетворяющих фильтру (+ пагинация) (протестировать на операторах Where, OrderBy, GroupBy, Take, посмотреть sql, в логах(консоли), оценить отличие от Linq ;

Самостоятельно реализовать аналогичный подход в «EmployeeService».

Провести проверку работы готового сервиса в теле теста. Для добавления новых клиентов пользуемся ранее реализованным «TestDataGenerator».

Вопросы для самоконтроля:

- Какую роль выполняет EF?
- В каком виде запросы EF доходят до базы?
- Что представляет из себя режим отслеживания в EF?
- В чем отличие IEnumerable от IQueryable?
- Чем отличается ленивая загрузка от жадной?

Ресурсы:

- [Entity Framework Core | Введение \(metanit.com\)](https://metanit.com/core/entity-framework/)

- [Entity Framework Core | PostgreSQL \(metanit.com\)](#)
- [Entity Framework Core | Управление схемой БД и миграции \(metanit.com\)](#)
- [Entity Framework Core | Конфигурация подключения \(metanit.com\)](#)
- [Entity Framework Core | Модели, Fluent API и аннотации данных \(metanit.com\)](#)
- [IEnumerable и IQueryable | Entity Framework 6 \(metanit.com\)](#)
- [IQueryable Интерфейс \(System.Linq\) | Microsoft Docs](#)

IDisposable. Ключевое слово using

При работе с платформой .NET мы периодически сталкиваемся с термином "[управляемый код](#)". Что это такое? Управляемым кодом называется код, выполнение которого управляется средой выполнения. В этом случае соответствующая среда выполнения называется общезыковой средой выполнения именуемой CLR. [Среда CLR](#) отвечает за использование управляемого кода, его компиляцию в машинный код и последующее выполнение. Кроме того, среда выполнения предоставляет несколько важных служб, например, автоматическое управление памятью. За последнее в dotnet отвечает [сборщик мусора](#) — служба, которая автоматически высвобождает неиспользуемую память.

Теперь поговорим о неуправляемом коде и неуправляемых ресурсах.

В мире неуправляемого кода практически за все отвечает программист. Сама программа представляет собой двоичный файл, который операционная система (ОС) загружает в память и запускает. За все остальное — от управления памятью до различных аспектов безопасности — отвечает программист.

Среда CLR позволяет пересекать границы между управляемым и неуправляемым кодом.

Это называется межпрограммным взаимодействием и позволяет нам, например, заключить неуправляемую библиотеку (написанную, например на C++) в оболочку и вызвать ее.

Основным типом неуправляемых ресурсов являются объекты, образующие обертку для ресурсов операционной системы, таких как дескриптор файлов, дескриптор окна или сетевое подключение.

То есть неуправляемым ресурсом является выделенная память, за управление которой ответственна не среда CLR, а сам программист.

Рассмотрим два способа, с помощью которых C# разработчик может работать с неуправляемыми ресурсами.

IDisposable и финализаторы — инструменты, с помощью которых мы можем [освободить ресурсы](#), с которыми работаем, чтобы они стали доступны для других задач.

IDisposable — это шаблон проектирования, предоставляемый dotnet для управления неуправляемыми ресурсами.

Он позволяет разработчику сказать: "Мне больше не нужны ресурсы, которые использует этот объект. Освободи их прямо сейчас. Сделай это сейчас, потому что эти ресурсы ограничены. Они больше не нужны мне для текущей задачи."

Рассмотрим, почему важно освобождать ресурсы, на примере подключений к базе данных postgres.

Получим подключение:

```
private NpgsqlConnection GetConnection()
{
    // Получаем строку подключения из конфигурации
    var connectionString =
    ConfigurationManager.ConnectionStrings["Postgres"].ConnectionString;

    //Создаем соединение
    var connection = new NpgsqlConnection(connectionString);
    return connection;
}
```

Откроем подключение:

```
private void OpenConnection()
{
    var connection = GetConnection();
    connection.Open(); // Открываем подключение
    LogOpenedConnectionCount(); // Записываем число открытых
                                // подключений
}

private void LogOpenedConnectionCount()
{
    _connectionsCounter++;
    Console.WriteLine(_connectionsCounter);
}
```

Сделаем попытку открыть 200 подключений к postgres:

```
public void StartOpenConnections()
{
    for (var i = 0; i < 200; i++)
    {
        OpenConnection();
    }
}
```

И получим исключение: **Unhandled exception. Npgsql.NpgsqlException (0x80004005): The connection pool has been exhausted, either raise 'Max Pool Size' (currently 100) or 'Timeout' (currently 15 seconds) in your connection string.**

В котором сообщается о том, что мы исчерпали все доступные соединения (по умолчанию их 100). [Подробнее о максимальном числе соединений к базе Postgres.](#)

Как мы можем исправить данную ситуацию? Правильный ответ — грамотно использовать ресурсы.

Для начала давайте внимательно посмотрим на класс NpgsqlConnection:

```
public sealed class NpgsqlConnection : DbConnection, ICloneable,
    IComponent
{...}
```

Как видим, он наследуется от класса DbConnection:

```
public abstract class DbConnection : Component, IDbConnection,
    IDisposable, IAsyncDisposable
{...}
```

Который реализует интерфейс IDisposable:

```
public interface IDisposable
{
    void Dispose();
}
```

Это простой интерфейс с единственным методом `Dispose`. Его предполагаемое назначение в том, что при вызове, он должен освобождать свои ресурсы. Для класса `NpgsqlConnection` метод `Dispose` уже реализован. Он закрывает соединение с базой данных, освобождая ценный ограниченный ресурс — соединение.

Применим метод `Dispose` в нашем случае:

```
private void OpenConnection()
{
    var connection = GetConnection();
    connection.Open(); // Открываем подключение
    LogOpenedConnectionCount(); // Записываем число открытых
                                // подключений
    DoWork(connection); // используем соединение в своих целях,
                        // например, вызываем запрос к бд
    connection.Dispose(); // Вызовем реализацию метода Dispose
                          // освободив соединение
}
```

И в этом случае наше приложение завершается без ошибки.

Один из выводов, которые можно сделать из этого примера — если вы используете `IDisposable` класс, воспринимайте это как сигнал о том, что этот объект имеет некоторые ценные ресурсы, которые сами по себе не освободятся, и как только вы закончите работу с этим объектом, вы должны вызвать метод `Dispose`.

Вызов метода `Dispose` вручную действительно работает, но есть вариант сделать это лучше. Использовать ключевое слово `using`. Почему этот вариант лучше? При использовании `using` происходит “закулисная” обработка исключений, что гарантирует нам, что `Dispose` будет вызван, даже если перед этим возникнет исключение.

Что значит “закулисная” обработка? Результата, аналогичного использованию конструкции `using` можно добиться, поместив объект и работу с ним внутрь `try-finally` блока, а затем вызвав `Dispose` (или `DisposeAsync`) в `finally` блоке. На самом деле именно так конструкции `using` транслируются компилятором.

С помощью оператора `using` можно создать один или несколько экземпляров `IDisposable` объектов, а затем в блоке кода определить область использования их ресурсов.

```
private void OpenConnection()
{
    using (var connection = GetConnection())
    {
        connection.Open();
        LogOpenedConnectionCount();
    } // как только пройдем закрытую фигурную скобку будет вызван
    //метод Dispose
}
```

Коротко о [финализаторах](#).

Финализатор в C# — необязательный метод, который, если он реализован, будет выполняться как часть процесса сборки мусора. Причина, по которой финализатор нужен для неуправляемых ресурсов заключается в том, что Garbage Collector не знает о неуправляемых ресурсах и никогда их не освободит, даже если вы реализуете метод Dispose, который правильно освобождает ваши ресурсы. Поэтому финализатор — ваша страховочная сетка, последний шанс освободить неуправляемые ресурсы. Если класс не владеет неуправляемыми ресурсами, скорее всего финализатор вам не нужен. В рамках данного урока мы не будем углубляться в тему финализаторов. Более подробно эту тему можно изучить [здесь](#).

Теперь вернемся к паттерну IDisposable и реализуем его на конкретном примере.

Пусть у нас есть класс, который владеет двумя типами неуправляемых ресурсов: соединением с базой данных и фрагментом неуправляемой памяти (за управление которой ответственна не среда CLR, а сам программист).

```
public class ConnectionAndMemory
{
    public static long TotalFreed { get; private set; }
    public static long TotalAllocated { get; private set; }

    private NpgsqlConnection _connection;
    private IntPtr _chunkHandle; // адрес
                                // в неуправляемой памяти
    private int _chunkSize; // число выделенных байтов
}
```

```

public ConnectionAndMemory(int chunkSize)
{
    _connection = Demo.GetConnection();
    _connection.Open();

    _chunkSize = chunkSize;

    // Выделяем память из неуправляемой памяти процесса
    _chunkHandle = Marshal.AllocHGlobal(chunkSize);

    TotalAllocated += chunkSize;
}

public void DoWork() { } // Фиктивный метод. Подразумевается, что
//здесь вы работаете с ресурсами.
}

```

Сейчас наш класс не реализует паттерн IDisposable.

Посмотрим, что произойдет, если мы будем работать с таким классом.

Напишем метод, который создает экземпляры нашего класса:

```

public void CreateConnectionsAndMemory(int count)
{
    var random = new Random();
    for (int i = 0; i < count; i++)
    {
        var chunkSize = random.Next(4096);
        var connectionAndMemory = new
ConnectionAndMemory(chunkSize);
        connectionAndMemory.DoWork();
    }
}

```

И посмотрим, сколько памяти было выделено, а сколько освобождено.

```

CreateConnectionsAndMemory(100);
Console.WriteLine($"Total Allocated:
{ConnectionAndMemory.TotalAllocated}");
Console.WriteLine($"Total Freed: {ConnectionAndMemory.TotalFreed}");

```

Вывод консоли:

Total Allocated: 121017

Total Freed: 0

Соответственно при запуске метода `CreateConnectionsAndMemory` с параметром более, чем 100, мы бы получили уже известную ошибку от postgres.

В нынешнем виде у нас плохое управление ресурсами. Поскольку ничего не реализовано для освобождения ресурсов. Исправим это. Реализуем `IDisposable` паттерн.

Начнем с метода для класса `CreateConnectionsAndMemory`, освобождающего неуправляемую память.

Здесь важно подчеркнуть то, что метод `Dispose` должен быть идемпотентным, то есть устойчивым к многократным вызовам — это рекомендация microsoft.

Помня об этом, добавим в класс приватное поле типа `bool _isFreed`, которое сообщает нам о том, была ли освобождена неуправляемая память.

```
private void ReleaseUnmanagedResources()
{
    if (_isFreed) return;

    Marshal.FreeHGlobal(_chunkHandle);
    TotalFreed += chunkSize;
    _isFreed = true;
}
```

Теперь реализуем и сам метод `Dispose`:

```
protected virtual void Dispose(bool disposing)
{
    ReleaseUnmanagedResources();
    if (disposing)
    {
        _connection.Dispose(); // уже идемпотентен
    }
}

public void Dispose()
{
    Dispose(true);
}
```

Конечный вариант нашего класса будет иметь следующий вид:

```
public class ConnectionAndMemory : IDisposable
{
    public static long TotalFreed { get; private set; }
    public static long TotalAllocated { get; private set; }

    private NpgsqlConnection _connection;
    private IntPtr _chunkHandle;
    private int _chunkSize;
    private bool _isFreed;

    public ConnectionAndMemory(int chunkSize)
    {
        _connection = Demo.GetConnection();
        _connection.Open();

        _chunkSize = chunkSize;
        _chunkHandle = Marshal.AllocHGlobal(chunkSize);
        TotalAllocated += chunkSize;
    }

    private void ReleaseUnmanagedResources()
    {
        if (_isFreed) return;

        Marshal.FreeHGlobal(_chunkHandle);
        TotalFreed += _chunkSize;
        _isFreed = true;
    }

    public void DoWork() { } // Фиктивный метод. Подразумевается,
    что здесь вы работаете с ресурсами.

    protected virtual void Dispose(bool disposing)
    {
        ReleaseUnmanagedResources();
        if (disposing)
        {
            _connection?.Dispose();
        }
    }

    public void Dispose()
    {
        Dispose(true);
    }
}
```

Очевидный вопрос, который может возникнуть — почему два метода `Dispose`.

Ключ к пониманию состоит в том, чтобы понять мотив создания `protected virtual` метода `Dispose`. Этот метод нужен нам по двум причинам:

1. Мы объединили освобождение нескольких ресурсов: соединения с базой данных и неуправляемой памяти в один метод. Это хорошо, поскольку он позволяет нам легко увидеть, не забыли ли мы что-либо очистить.
2. Что более важно — виртуальность. Мы можем реализовать иерархию классов и позволить каждому уровню иерархии освободить ресурсы, принадлежащие этому конкретному уровню. Поэтому, если мы например, унаследуем, новый класс от `ConnectionAndMemory`, где будет использован новый ресурс. Пусть дескриптор файла, то наследующий класс может просто сделать `override` этого метода. Освободить свои ресурсы, а потом вызвать базовую реализацию.

Теперь посмотрим на публичный метод `Dispose`.

Он сведен к минимуму — вызываем реализацию `protected Dispose(true)`. Почему `true`, а не `false` и зачем нам вообще параметр в методе `Dispose`?

Вызов `Dispose(false)` понадобился бы нам в финализаторе.

Идея здесь состоит в том, чтобы `Dispose(bool disposing)` знал, вызывается ли он для выполнения явной очистки — `Dispose(true)` или вызывается из-за сборки мусора — `Dispose(false)`. Это различие полезно, потому что при удалении явно, метод `Dispose(bool disposing)` может безопасно выполнять код, используя поля ссылочного типа, которые ссылаются на другие объекты, зная наверняка, что эти другие объекты еще не завершены или не удалены.

Когда `disposing` равно `false`, метод `Dispose(bool disposing)` не должен выполнять код, ссылающийся на поля ссылочного типа, так как эти объекты уже могут быть завершены.

Больше информации по реализации `IDisposable` паттерна можно найти [здесь](#).

Вернемся к примеру.

Всё, что нам остаётся сделать, это добавить using:

```
public void CreateConnectionsAndMemory(int count)
{
    var random = new Random();
    for (int i = 0; i < count; i++)
    {
        var chunkSize = random.Next(4096);
        using (var connectionAndMemory = new
ConnectionAndMemory(chunkSize))
        {
            connectionAndMemory.DoWork();
        }
    }
}
```

Запустив приложение, убедимся, что ресурсы очищаются корректно.

Вывод консоли:

Total Allocated: 202369

Total Freed: 202369

Практическая часть:

(проходит совместно с теоретической)

Вопросы для самоконтроля:

- Что такое неуправляемый ресурс?
- Что такое сборщик мусора?
- Для чего нужен интерфейс IDisposable?
- Что такое финализатор?
- Какую работу выполняет оператор using?
- Можно ли продолжить работу с объектом после вызова Dispose?

Stream. FileStream

Поток - это абстракция последовательности байтов, например файла, устройства ввода-вывода, межпроцессного канала связи или сокета TCP/IP. Класс **Stream** и его производные классы предоставляют общее представление этих различных типов входных и выходных данных и изолируют программиста от конкретных деталей операционной системы и базовых устройств.

В основе потоковой архитектуры .NET лежат три понятия:

- опорное хранилище (backing store)
- декоратор (decorator)
- адаптер (adapter)

Опорное хранилище — это конечная точка ввода-вывода: файл, сетевое подключение и т.д. Оно может представлять собой либо источник, из которого последовательно считываются байты, либо приемник, куда байты последовательно записываются, либо и то и другое вместе.

Чтобы использовать опорное хранилище его нужно открыть. Этой цели и служат потоки, которые в .NET представлены классом **System.IO.Stream**, содержащий методы для чтения, записи и позиционирования потоков.

Потоки не загружают опорное хранилище в память целиком, а читают его последовательно по байтам либо блокам управляемого размера. Поэтому поток может потреблять мало памяти независимо от размера его опорного хранилища.

Потоки делятся на две категории:

- потоки опорных хранилищ — потоки, жестко привязанные к конкретным типам опорных хранилищ, такие как **FileStream** или **NetworkStream**
- потоки-декораторы — наполняют другие потоки, трансформируя данные тем или иным способом, такие как **DeflateStream** или **CryptoStream**

Декораторы освобождают потоки опорных хранилищ от необходимости самостоятельно реализовывать такие вещи, как сжатие и шифрование. Декораторы можно подключать во время выполнения, а также соединять их в цепочки (т.е. использовать несколько декораторов в одном потоке).

Потоки включают три основные операции: чтение, запись, в некоторых случаях поиск.

По окончании использования потока, выделенную ему память следует прямо или косвенно освободить. Чтобы сделать это, вызовите его метод **Dispose** в блоке **try/finally**. или используйте языковые конструкции, такие как **using**.

Dispose освобождает ресурсы операционной системы, такие как дескрипторы файлов, сетевые подключения или память, используемые для любой внутренней буферизации.

Некоторые из наиболее часто используемых потоков, наследующих от **Stream** – **FileStream**, **MemoryStream**, **NetworkStream**.

1. Работа с FileStream

Рассмотрим работу с потоками на примере класса **FileStream**.

Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Для создания объекта **FileStream** можно использовать как конструкторы этого класса, так и статические методы класса **File**. Конструктор **FileStream** имеет множество перегруженных версий, из которых отметим лишь одну, самую простую и используемую:

FileStream(string filename, FileMode mode)

Здесь в конструктор передается два параметра: путь к файлу и перечисление **FileMode** указывающее на режим доступа к файлу.

FileMode.Create – указывает, что если файла с указанным именем не существует, он будет создан. В случае, если такой файл уже существует, он будет перезаписан.

FileMode.Open – указывает, что операционная система должна открыть уже существующий файл. Если такого файла не существует, при создании экземпляра **FileSrteam** будет сгенерировано исключение.

FileMode.Append в случае, если файл существует, находит его конец и пишет новые записи туда, дополняя уже записанную там информацию. В случае, если файла нет, создает его.

Более подробную информацию по **FileMode** вы можете прочитать по ссылке: [FileMode Перечисление \(System.IO\) | Microsoft Docs](#)

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Давайте разберем на практике. Создадим класс **StreamRepository**, который будет содержать методы чтения и записи в файл. Так как мы планируем, что читать и писать этот класс будет в один и тот же файл, то передаём ему путь и имя файла один раз при создании через конструктор:

```
public class StreamRepository
{
    private string _pathToDirectory { get; set; }
    private string _textFileName { get; set; }

    public StreamRepository(string pathToDirectory, string textFileName)
    {
        _pathToDirectory = pathToDirectory;
        _textFileName = textFileName;
    }
}
```

Добавим в этот класс метод **WriteFileFromString**, который будет принимать в себя строку и затем записывать её в текстовый файл через **FileStream**:

```
public void WriteFileFromString(string text)
{
    DirectoryInfo dirInfo = new DirectoryInfo(_pathToDirectory);
    // Проверяем, есть ли по указанному пути папка
    if (!dirInfo.Exists)
    {
        // Если папки нет - создаём
        dirInfo.Create();
    }
    string fullPath = Path.Combine(_pathToDirectory, _textFileName);
    using (FileStream fileStream = new FileStream(fullPath,
        FileMode.OpenOrCreate))
    {
        // Преобразуем строку в байты
        byte[] array = System.Text.Encoding.Default.GetBytes(text);
        // Запись массива байтов в файл
    }
}
```

```
        filestream.Write(array, 0, array.Length);  
        Console.WriteLine("Текст записан в файл");  
    }  
}
```

Теперь добавим в в этот класс метод для чтения файла, который будет считывать текстовый файл и возвращать строку:

```
public string ReadFileToString()  
{  
    string fullPath = Path.Combine(_pathToDirectory, _textFileName);  
    using (FileStream fileStream = File.OpenRead(fullPath))  
    {  
        // Резервируем массив для хранения данных  
        byte[] array = new byte[fileStream.Length];  
        // Считываем данные  
        fileStream.Read(array, 0, array.Length);  
        // Декодируем байты в строку  
        string textFromFile = System.Text.Encoding.Default  
            .GetString(array);  
        return textFromFile;  
    }  
}
```

Сразу видим что код **Path.Combine(_pathToDirectory, _textFileName)** дублируется в разных методах одного класса, поэтому выносим в отдельный приватный метод:

```
private string GetFullPathToFile(string pathToFile, string fileName)  
{  
    return Path.Combine(pathToFile, fileName);  
}
```

И изменяем получение пути в методах чтения и записи на использование этого метода. Теперь давайте реализуем эти методы:


```

static void Main(string[] args)
{
    string pathToDirectory = Path.Combine("C:", "TestFiles");
    string fileName = "note.txt";

    StreamRepository streamRepository = new
    StreamRepository(pathToDirectory, fileName);

    Console.WriteLine("Введите строку для записи в файл");

    string text = Console.ReadLine();

    streamRepository.WriteFileFromString(text);

    string textFromFile = streamRepository.ReadFileToString();

    Console.WriteLine($"Текст из файла: {textFromFile}");
    Console.ReadLine();
}

```

Разберем этот пример.

Вначале мы создаем строку – путь к папке (далее – каталогу), в которую мы будем сохранять наш файл. Делаем это через статический метод **Combine** класса **Path**. Его использование удобнее, чем написание вручную **"C:\\TestFiles"** и ограждает нас от ошибок, в случае, если нам нужно составить путь из **pathToDirectory** и **fileName**, нам не нужно вспоминать и проверять, есть у нас в конце значения **pathToDirectory** **"\"** или нет.

Затем, в методе **WriteFileFromString** мы создаем объект класса **DirectoryInfo**, который предоставляет нам методы для работы над каталогом, путь к которому мы передали в конструктор к этому объекту. Через него мы проверяем, есть ли по указанному пути каталог с именем **«TestFiles»**, если его нет, создаем.

Получаем текст, который будем писать в файл и определяем имя файла, в который этот текст будет записан. Составляем полный путь к файлу из **pathToDirectory** и **filename**, потому что путь, который принимает конструктор **FileStream**, должен вести к самому файлу, с которым вы хотите взаимодействовать.

И при чтении, и при записи используется оператор **using**. Не надо путать данный оператор с директивой **using**, которая подключает пространства

имен в начале файла кода. Оператор `using` определяет область кода, в конце которой, вызывается метод `Dispose` у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная `fileStream`.

И при записи, и при чтении применяется объект кодировки `Encoding.Default` из пространства имен `System.Text`. В данном случае мы используем два его метода: `GetBytes` для получения массива байтов из строки и `GetString` для получения строки из массива байтов.

2. Работа с `StreamReader/StreamWriter`

В итоге введенная нами строка записывается в файл `note.txt`. По сути это бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

```
fileStream.WriteByte(13);  
fileStream.WriteByte(183);
```

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - `StreamReader` и `StreamWriter`. Давайте изменим написанные нами методы чтения и записи файла с использованием этих классов:

```
public void WriteFileFromString(string text)  
{  
    DirectoryInfo dirInfo = new DirectoryInfo(_pathToDirectory);  
    // Проверяем, есть ли по указанному пути папка  
    if (!dirInfo.Exists)  
    {  
        // Если папки нет - создаём  
        dirInfo.Create();  
    }  
    string fullPath = GetFullPathToFile(_pathToDirectory, _textFileName);  
    using (FileStream fileStream = new FileStream(fullPath,  
        FileMode.OpenOrCreate))  
    {  
        using (StreamWriter streamWriter = new StreamWriter(fileStream))  
        {  
            streamWriter.Write(text);  
        }  
        Console.WriteLine("Текст записан в файл");  
    }  
}
```

```

public string ReadFileToString()
{
    string fullPath = GetFullPathToFile(_pathToDirectory, _textFileName);
    using (FileStream fileStream = File.OpenRead(fullPath))
    {
        using (StreamReader streamReader = new StreamReader(fileStream))
        {
            string textFromFile = streamReader.ReadToEnd();
            return textFromFile;
        }
    }
}

```

Другой способ создания объекта `FileStream` представляют статические методы класса `File`:

```

FileStream File.Create(string path);
FileStream File.Open(string path, FileMode mode);
FileStream File.OpenRead(string path);
FileStream File.OpenWrite(string path);
FileStream File.Delete(string path);

```

3. Работа с CSV файлами

Давайте представим, что нам нужно сохранить объект в файл, с сохранением возможности потом считать данные из файла обратно в объект. На данном этапе формат `.txt` файла нам не подойдёт. Тут нам на помощь приходит формат `.csv`, который позволит сохранить данные в формате таблицы, где столбцы будут свойства объекта, а каждая новая строка – объект. Для того, чтобы экспортировать объект в формат `.csv`, нам необходим nuget пакет `CsvHelper`. Установим в наш solution этот пакет версии 27.2.1. Теперь нам стали доступны классы `CsvWriter` и `CsvReader`, через которые мы и сможем сохранять данные в формат `.csv` и затем считывать их. Давайте создадим новый класс `PersonExporter` и реализуем в его рамках методы чтения и записи в CSV.

```

public class PersonExporter
{
    private string _pathToDirecory { get; set; }
    private string _csvFileName { get; set; }

    public PersonExporter(string pathToDirectory, string csvFileName)
    {
        _pathToDirecory = pathToDirectory;
        _csvFileName = csvFileName;
    }
}

```

```

public void WritePersonToCsv(Person person)
{
    // Создаём каталог для файла
    DirectoryInfo dirInfo = new DirectoryInfo(_pathToDirecory);
    if (!dirInfo.Exists)
    {
        dirInfo.Create();
    }
    string fullPath = GetFullPathToFile(_pathToDirecory, _csvFileName);
    using (FileStream fileStream = new FileStream(fullPath,
        FileMode.OpenOrCreate))
    {
        using (StreamWriter streamWriter = new StreamWriter(fileStream))
        {
            using (var writer = new CsvWriter(streamWriter,
                CultureInfo.InvariantCulture))
            {
                // Формируем заголовки будущей таблицы
                writer.WriteField(nameof(Person.Name));
                writer.WriteField(nameof(Person.Age));

                // Переход на следующую строку таблицы
                writer.NextRecord();

                writer.WriteField(person.Name);
                writer.WriteField(person.Age);

                writer.NextRecord();

                // Очищает буфер, который был задействован CsvWriter-ом
                writer.Flush();
            }
        }
    }
}

```

```

public Person ReadPersonFromCsv()
{
    string fullPath = GetFullPathToFile(_pathToDirecory, _csvFileName);
    using (FileStream fileStream = new FileStream(fullPath,
        FileMode.OpenOrCreate))
    {
        using (StreamReader streamReader = new StreamReader(fileStream))
        {
            using (var reader = new CsvReader(streamReader,
                CultureInfo.InvariantCulture))
            {
                // Считываем из файла данные объекта Person
                reader.Read();
                Person readPerson = reader.GetRecord<Person>();
            }
        }
    }

    Console.ReadLine();
}

```

```

private string GetFullPathToFile(string pathToFile, string fileName)
{
    return Path.Combine(pathToFile, fileName);
}

```

Теперь реализуем методы класса PersonExporter:

```

static void Main(string[] args)
{
    string pathToDirectory = Path.Combine("C:", "TestFiles");
    string fileName = "note.csv";
    PersonExporter personExporter = new
    PersonExporter(pathToDirectory, fileName);

    Person person = new Person()
    {
        Name = "John",
        Age = 30
    };

    personExporter.WritePersonToCsv(person);
    Person personFromFile = personExporter.ReadPersonFomCsv();

    Console.WriteLine($"Данные персоны из файла. Имя:
    {personFromFile.Name}, Возраст: {personFromFile.Age}");
    Console.ReadLine();
}

```

В результате выполнения этого кода мы получим следующую таблицу:

	A	B
1	Name, Age	
2	John, 30	
3		
4		

Практическая часть:

- Добавьте в ваше решение новый проект по шаблону библиотека классов, назовите его “ExportTool”. В эту библиотеку добавьте новый класс “ExportService” и реализуйте в нём метод экспорта клиентских данных из базы в файл CSV. Метод принимает на вход коллекцию клиентов банка;
- Реализуйте метод считывания файла и добавления клиентских данных в базу.

Вопросы для самоконтроля:

- Зачем нужен оператор using при работе с файлами?
- Зачем нужен StreamWriter при уже открытом потоке FileStream?
- В случае, если мы выбираем FileMode.Open при создании объекта FileStream, зачем нам проверка через DirectoryInfo о наличии файла?

Ресурсы:

- [Stream Класс \(System.IO\) | Microsoft Docs](#)
- [NetworkStream и текстовые потоки в C# и .NET \(metanit.com\)](#)
- [FileStream Класс \(System.IO\) | Microsoft Docs](#)
- [C# и .NET | FileStream. Чтение и запись файла \(metanit.com\)](#)
- <https://joshclose.github.io/CsvHelper/getting-started/>

Расширения, Extensions

Методы расширения позволяют "добавлять" методы в существующие типы без создания нового производного типа, повторной компиляции и иного изменения первоначального типа. Методы расширения представляют собой особую разновидность статического метода, но вызываются так же, как методы экземпляра в расширенном типе. Для клиентского кода нет видимого различия между вызовом метода расширения и вызовом методов, фактически определенных в типе.

Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Или если мы не можем использовать стандартный механизм наследования, например, если классы определены с модификатором `sealed`.

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. Метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: `this имя_типа название_параметра`.

Следующий пример показывает метод расширения, определенный для класса `System.String`

Пример метода расширения типа `System.String`. Метод `WordCount` принимает на вход строку, возвращает число слов: число подстрок, разделенных пробелом, точкой или вопросительным знаком:

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        if (str == null) throw new
ArgumentNullException(nameof(str));

        return str.Split(new char[] { ' ', '.', '?', '!' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

В статическом классе `StringExtensions` мы определили метод расширения типа `System.String`. Поэтому в методе `WordCount` первый параметр определен как `this string str`. Так как наш метод будет относиться к типу `string`, то мы и используем данный тип.

Затем у всех строк мы можем вызвать данный метод. Причем нам уже не надо указывать первый параметр. Значения для остальных параметров, если такие имеются, передаются в обычном порядке.

Пример использования метода расширения:

```
using System;
using ExtensionMethods;

public class Program
{
    static void Main(string[] args)
    {
        string s = "Привет мир";
        int i = s.WordCount();
        Console.WriteLine(i);

        Console.ReadLine();
    }
}
```

В С# метод расширения имеет доступ только к публичным членам класса. Другим ограничением является то, что если есть и встроенный метод, и расширение, приоритет даётся встроенному методу.

Самостоятельно:

- Порядок объявления и использования методов расширения
- Ограничения в реализации и использовании методов расширения
- Реализовать методы расширения для класса `int`, для операций над `TimeSpan`. Например `1,Seconds() = (TimeSpan) 00.00.01`;

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/how-to-implement-and-call-a-custom-extension-method>
- <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/how-to-create-a-new-method-for-an-enumeration>
- <https://metanit.com/sharp/tutorial/3.18.php>
- https://ru.wikipedia.org/wiki/Метод_расширения

Reflection - рефлексия, метаданные классов

Рефлексия - это механизм извлечения информации об управляемых типах, сборках и прочих атрибутах, путем обработки метаданных во время исполнения программы (runtime).

Представим себе такой метод

```
void PrintObjectProperties(object target);
```

в его задачу входит распечатать через запятую все публичные свойства объекта переданного в аргументы. Если бы мы заранее знали какого типа объекты нам передают то нет проблем, да и мы бы могли

переписать наш метод например так `void PrintObjectProperties(CustomType target);` но мы не знаем, вот тут приходит рефлексия на помощь.

Пример:

```
public string PrintObjectProperties(object target)
{
    if (target == null) throw new ArgumentNullException(nameof(target));

    var type = target.GetType(); // получаем тип
    var props = type.GetProperties(BindingFlags.Public |
    BindingFlags.Instance); // получаем все публичные свойства, не
    статические

    var values = props.Select(x => $"{x.Name} : {x.GetValue(target)}");
    // перебираем все свойства и описываем формат сохранения в строку

    return string.Join(", ", values); // формируем строку
}
```

Мы привели пример кода, который будет формировать строку из всех публичных свойств переданного объекта. Это достаточно универсальный механизм, но нужно помнить что делать такие операции массово не стоит это достаточно медленно.

В качестве области использования могу назвать - сериализаторы (serializer), мапперы (mapper).

Самостоятельно:

- Научиться создавать экземпляр класса по строковому наименованию, берем из прошлых занятий например "Triangle"
- Вызвать метод с параметрами
- Попробуйте получить закрытые свойства класса, считать из них значения

Ресурсы:

- [GetFields\(BindingFlags\)](#)
- [GetProperties\(BindingFlags\)](#)
- [GetMethods\(BindingFlags\)](#)
- [GetConstructors\(BindingFlags\)](#)

[Полный пример](#)

Serialization (Json, XML, Binary)

Сериализация - процесс перевода какой-либо структуры данных в форму удобную для сохранения или передачи (байты или текст).

Десериализация - восстановление начального состояния структуры данных из сериализованной формы. Процесс обратный сериализации.

Стандартные сериализаторы .net в основном работают с атрибутом `Serializable`, данным атрибутом необходимо пометить класс, который будет сериализован. Если класс не будет помечен атрибутом, то при сериализации будет исключение `SerializationException`. Если какое то поле из класса требуется проигнорировать при сериализации, то его можно пометить атрибутом `NonSerialized`. Атрибут `Serializable` не наследуется.

В .net, на данный момент, поддерживаются такие форматы как:

1. Бинарный - `BinaryFormatter`
2. SOAP - `SoapFormatter`
3. XML - `XmlSerializer`
4. JSON - `DataContractJsonSerializer`, `newtonsoft`

Рассмотрим сериализацию на примере класса пользователя:

```
[Serializable]
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }

    public int Age {
        get
        {
            DateTime dateNow = DateTime.Now;
            int year = dateNow.Year - DateOfBirth.Year;
            if (dateNow.Month < dateBirthDay.Month ||
                (dateNow.Month == dateBirthDay.Month && dateNow.Day
< dateBirthDay.Day))
            {
                year--;
            }
            return year;
        }
    }
}
```

У данного пользователя есть 3 свойства, которые будут сериализованы, и одно вычисляемое свойство, которое не попадёт в сериализованные данные.

Создадим несколько пользователей, для дальнейшей работы с ними:

```
User user1 = new User {Id = 1, Name = "Tom", DateOfBirth = new
DateTime(1987, 2, 25)};

User user2 = new User {Id = 2, Name = "Bill", DateOfBirth = new
DateTime(1990, 5, 14)};

User[] users = new User[] { user1, user2 };
```

1. Бинарная сериализация

Пример кода бинарной сериализации:

```
BinaryFormatter formatter = new BinaryFormatter();

// сериализация
using (FileStream fs = new FileStream("users.dat",
FileStream.OpenOrCreate))
{
    formatter.Serialize(fs, users); // сериализуем весь массив
}

// десериализация
User[] deserializeUsers;
using (FileStream fs = new FileStream("users.dat",
FileStream.OpenOrCreate))
{
    deserializeUsers = (User[]) formatter.Deserialize(fs);
}

// выводим данные
foreach (User u in deserializeUsers)
{
    Console.WriteLine("Имя: {0} --- Возраст: {1}", u.Name, u.Age);
}
```

Как показано на примере выше, мы можем сериализовать коллекцию экземпляров класса. Так же, результат сериализации, помещён в файл, на месте FileStream, может быть любой поток. При десериализации нам

нужно еще преобразовать объект, возвращаемый функцией `Deserialize`, к нужному типу.

2. Сериализация в XML

Сериализация в xml, очень похожа на бинарную сериализацию, но есть несколько отличий. самое главное из отличий, сериализуемый класс должен иметь конструктор без параметров. Так же сериализатор работает только с открытыми свойствами класса и сериализатор, при создании, требует тип сериализуемого объекта.

```
XmlSerializer formatter = new XmlSerializer(typeof(User[]));

// сериализация
using (FileStream fs = new FileStream("users.xml",
    FileMode.OpenOrCreate))
{
    formatter.Serialize(fs, users);
}

// десериализация
User[] deserilizeUsers;
using (FileStream fs = new FileStream("users.xml",
    FileMode.OpenOrCreate))
{
    deserilizeUsers = (User[] )formatter.Deserialize(fs);
}
```

Получаем результат:

```
<?xml version="1.0"?>
<ArrayOfUsers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <User>
    <Id>1</Id>
    <Name>Tom</Name>
    <DateOfBirth>1987-07-25T00:00:00</DateOfBirth>
  </User>
  <User>
    <Id>2</Id>
    <Name>Bill</Name>
    <DateOfBirth>1990-05-14T00:00:00</DateOfBirth>
  </User>
</ArrayOfUsers>
```

3. Сериализация в JSON

В отличие от приведенных выше сериализаторов, для json рассмотрим сторонний сериализатор newtonsoft. Он устанавливается через nuget.

```
string json = JsonConvert.SerializeObject(users);  
[  
  {  
    "Id": 1,  
    "Name": "Tom",  
    "DateOfBirth": "1987-07-25T00:00:00"  
  },  
  {  
    "Id": 2,  
    "Name": "Bill",  
    "DateOfBirth": "1990-05-14T00:00:00"  
  }  
]
```

Десериализация выглядит так же просто, как и сериализация:

```
User[] users = JsonConvert.DeserializeObject<User[]>(json);
```

Для работы данной библиотеки не требуется атрибут Serializable, а для того что бы свойство не было сериализовано, используется атрибут JsonIgnore.

Практическая часть:

В сервисе “ExportTool.ExportService” добавить методы для экспорта клиента в сериализованном виде в текстовый файл, и метод импорта клиента из сериализованной записи в файле. Применяем JSON сериализацию.

Вопросы для самоконтроля:

- Какие проблемы могут возникнуть при десериализации словаря?

- Что произойдёт при попытке сериализовать циклически связанный объект (имеет ссылку сам на себя, возможно через вложенные объекты)?

Самостоятельно:

- Реализовать сериализацию объекта со вложенными объектами (например банковский счет клиента)
- Изучить атрибуты newtonsoft

Ресурсы:

- [Сериализация | C# \(metanit.com\)](https://metanit.com/)

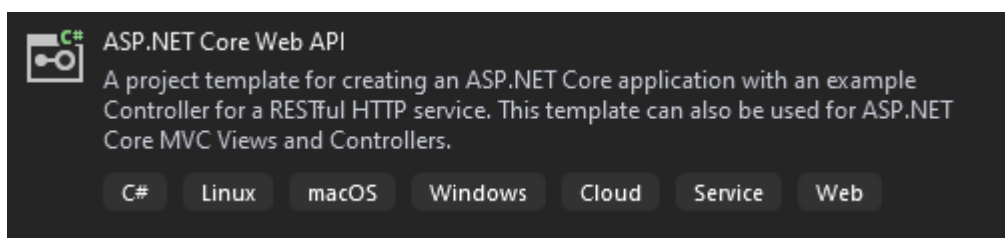
Asp.net controller

До сих пор мы вели нашу работу в рамках классов - сервисов. Однако для того чтобы предоставить доступ к написанному нами функционалу извне (например – команде, разрабатывающей фронтенд), нам необходима еще одна прослойка. Эту роль на себя берут веб приложения, центральным звеном которых являются контроллеры.

Контроллер является центральным звеном в архитектуре веб приложения и служит точкой входа для обращений извне. При получении запроса система маршрутизации выбирает для обработки запроса нужный контроллер и передает ему данные запроса. Контроллер обрабатывает эти данные и посылает обратно результат обработки.

В ASP.NET Core контроллер представляет обычный класс на языке C#.

Для ознакомления с контроллерами создадим новый проект, выбрав шаблон ASP.NET Core Web API



При этом будет сгенерирован проект, в котором уже присутствует рабочий контроллер «WeatherForecastController» с простейшим функционалом. Для поверхностного ознакомления достаточно обратить внимание на следующие моменты:

1) Имя класса строиться по следующему правилу – `ClassNameController`.

2) Класс наследуется от базового класса «`ControllerBase`»

```
3 references
public class WeatherForecastController : ControllerBase
{
```

3) Над классом расположены атрибуты `[ApiController]` и `[Route("[controller]")]`, первый из них не обязателен, а вот второй нам понадобится для того чтобы система маршрутизации смогла найти наш контроллер.

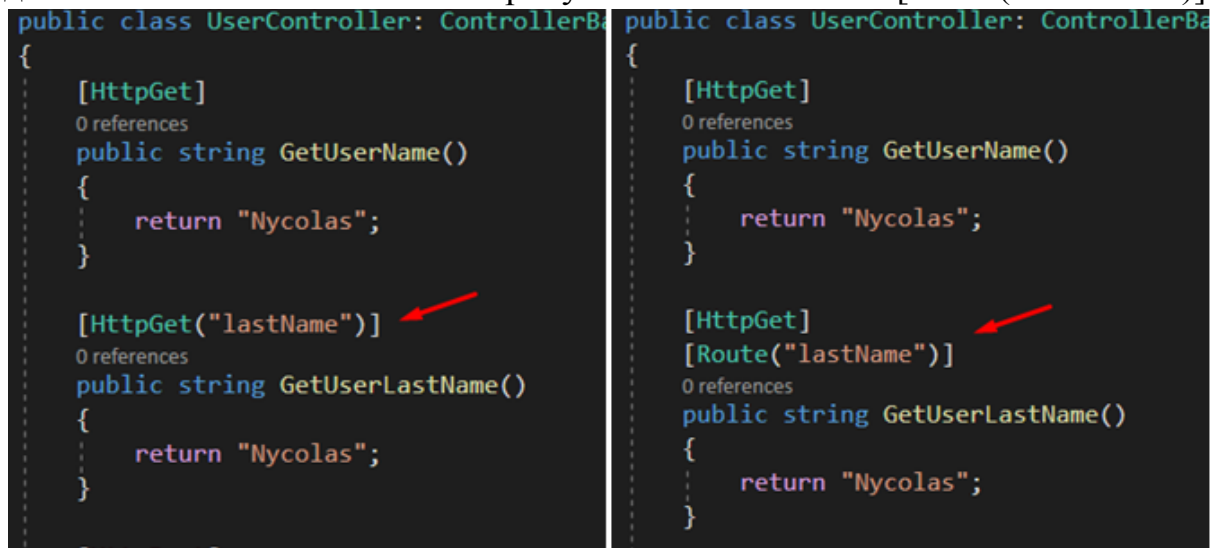
```
[ApiController]
[Route("[controller]")]
3 references
public class WeatherForecastController : ControllerBase
{
```


Подобрей про атрибуты можно почитать тут: [C# и .NET | Атрибуты \(metanit.com\)](#)

Про атрибуты маршрутизации тут: [ASP.NET Core | Атрибуты маршрутизации \(metanit.com\)](#)

4) Все публичные методы контроллера необходимо пометить специальным атрибутом, начинающимся с префикса “Http”. Атрибут выбирают в соответствии с функцией (запросом) который выполняет метод, чаще всего: [HttpGet] [HttpPost] [HttpPut] [HttpDelete], и др.

Атрибут служит идентификатором метода в системе маршрутизации, в связи с чем, не может быть двух методов с одинаковыми атрибутами. В определённых случаях, для конкретизации маршрута и преодоления проблем с дублированием, применяется атрибуты с параметрами либо добавляют атрибут [Route("lastName")]



```
public class UserController: ControllerBase
{
    [HttpGet]
    0 references
    public string GetUserName()
    {
        return "Nycolas";
    }

    [HttpGet("lastName")]
    0 references
    public string GetUserLastName()
    {
        return "Nycolas";
    }
}

public class UserController: ControllerBase
{
    [HttpGet]
    0 references
    public string GetUserName()
    {
        return "Nycolas";
    }

    [HttpGet]
    [Route("lastName")]
    0 references
    public string GetUserLastName()
    {
        return "Nycolas";
    }
}
```

Подобрей про систему маршрутизации можно почитать тут: [Маршрутизация к действиям контроллера в ASP.NET Core | Microsoft Docs](#)

Подобрей про контроллеры в ASP.NET Core Web API можно почитать тут: [ASP.NET Core | Web API \(metanit.com\)](#)

5) Вместе с контроллером «WeatherForecastController» шаблон содержит не знакомый нам пока класс «Startup». Данный класс является входной точкой в приложение ASP.NET Core. Этот класс производит конфигурацию приложения, настраивает сервисы, которые приложение будет использовать, устанавливает компоненты для обработки запроса.

В рамках поверхностного изучения достаточно упомянуть что в методах «ConfigureServices» и «Configure» класса Startup происходит конфигурация инструментария необходимого для корректной работы контроллеров. Например вызов методов services.AddControllers() или endpoints.MapControllers()

Подобней про класс «Startup» можно почитать тут: [ASP.NET Core | Класс Startup \(metanit.com\)](https://metanit.com/aspnet/core/11.html)

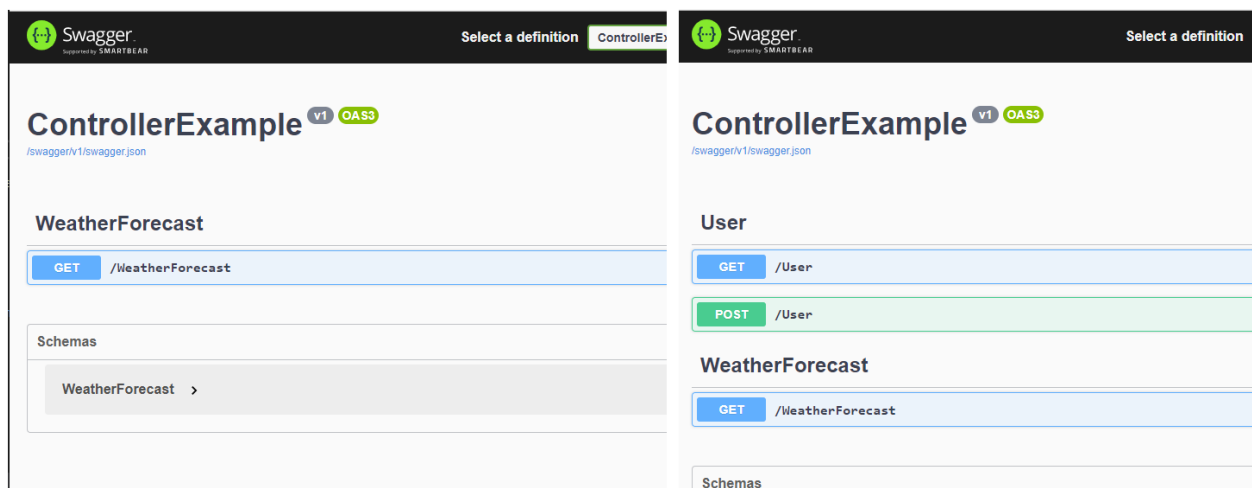
6) На данном этапе мы уже можем вызвать наш метод Get выполнив запрос в адресной строке браузера, используя следующий путь: `https://localhost:44328/ WeatherForecast`.

Однако, шаблон предоставляет доступ к более удобному инструменту обладающему более широким кругом возможностей. С этой целью в метод «ConfigureServices» класса «Startup» добавлена конфигурация для инструмента «Swagger».

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "ControllerExample", Version = "v1" });
    });
}
```

Данный инструмент автоматически строит графическую обложку для каждого метода, каждого контроллера, при условии что контроллер и методы помечены атрибутам, позволяющие системе маршрутизации идентифицировать их.

При запуске приложения, откроется окно в браузере в котором будет визуализирован единственный метод контроллера «WeatherForecast». Если мы добавим новый класс «UserController» и выполним настройки указанные в пунктах 1 – 5, то в окне генерируемом сваггером появиться



новые методы, нового контроллера:

Практическая часть:

В папку “API” добавить новый проект применив шаблон ASP.NET Core Web API.

В рамках нового проекта добавить контроллер для работы с клиентами банка.

В контроллере реализуем методы: получения клиента по идентификатору, добавления нового клиента, изменение клиента по идентификатору, удаления клиента, применив для каждого метода соответствующий атрибут.

Убедиться, что в окне Сваггера корректно отобразились все добавленные методы.

В теле созданных методов вызвать функционал ранее созданного сервиса «ClientService». С этой целью в конструкторе контроллера создаем экземпляр сервиса. Протестировать результат работы выполнив запросы через сваггер, в режиме отладки проследить путь выполнения программы.

Вопросы для самоконтроля:

- Какую роль играет контроллер в приложении?
- Допустимо ли применение двух методов с одинаковыми атрибутами маршрутизации?
- Что представляет собой инструмент «Swagger»?
- Что описывают атрибуты маршрутизации с префиксом HTTP?
- Какие ещё атрибуты маршрутизации с префиксом HTTP вы можете назвать?

Самостоятельно:

- Самостоятельно - реализовать контроллер для работы с сотрудниками.

Ресурсы:

- [ASP.NET Core | Web API \(metanit.com\)](https://metanit.com/)

Класс HttpClient и отправка запросов

HTTP - протокол, позволяющий получать различные ресурсы, например HTML-документы. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером (web-browser).

Объектом, над которым происходит работа протокола HTTP, является ресурс, на который указывает URI в запросе клиента. URI (Uniform Resource Identifier) – унифицированный идентификатор ресурса, простыми словами, это то, что указывается в строке браузера – имя запрашиваемого ресурса (страница, изображение, и т.д.).

В рамках текущей лекции мы проведем поверхностное ознакомление для начала работы с данным протоколом в рамках .Net. Подробнее о структуре HTTP запроса/ответа можно почитать в ресурсах прикрепленных к данной главе. Непременно изучите их.

1. Основные операции по HTTP

GET – запрос, используется для запроса содержимого указанного ресурса.

POST – запрос, применяется для передачи пользовательских данных заданному ресурсу (добавление новых данных на сервер).

PUT – запрос, применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существует ресурса, то сервер создаёт его и возвращает статус 201 (Created). Если же ресурс был изменен, то сервер возвращает 200 (Ok) или 204 (No Content) (обновление данных уже присутствующих на сервере).

DELETE – запрос, удаляет указанный ресурс.

2. Путь/маршрут (route/path), URI

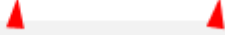
Унифицированный (единообразный) идентификатор ресурса. Последовательность символов, идентифицирующая абстрактный или физический ресурс.

Пример:

```
https://yandex.ru/api/users
```


Кроме того, в строке запроса могут передаваться параметры (query parameters). Параметры запроса указываются после знака “?” и разделяются с помощью знака “&”.

```
https://yandex.ru/api/users?userName=Petr&userAge=2
```



Пример передачи массива данных:

```
https://yandex.ru/api/users?ids=1&ids=2&ids=3|
```



Для простоты понимания, стоит упомянуть, что параметры запроса по сути являются параметрами, которые ожидаются на вход методами API к которому мы обращаемся. Ниже мы реализуем более наглядный пример.

3. Методы HttpClient

Для взаимодействия с веб-сервисами в .NET используется класс HttpClient из пространства имен System.Net.Http. С помощью его методов можно посылать определенный запрос к серверу.

HttpClient предоставляет следующие методы:

- GetAsync(): отправляет запрос типа Get
- DeleteAsync(): отправляет запрос типа Delete
- PostAsync(): отправляет запрос типа Post
- PutAsync(): отправляет запрос типа Put
- SendAsync(): отправляет запрос любого типа в зависимости от настроек заголовков запроса. Выше определенные методы являются частными случаями данного метода
- GetByteArrayAsync(): получает массив байтов в запросе типа Get
- GetStreamAsync(): получает поток Stream в запросе типа Get
- GetStringAsync(): получает строку в запросе типа Get

4. Отправка и получение данных

Данные запроса и ответа представлены в виде объекта класса `HttpContent`. Этот класс является абстрактным, и фактически мы будем работать с его производными классами:

- `StringContent`: применяется, если отправка или получение данных происходит в виде строки
- `ByteArrayContent`: применяется, если отправка или получение данных происходит в виде массива байтов
- `StreamContent`: применяется, если отправка или получение данных происходит, как правило, в виде файла, например, изображения

Если мы хотим получить данные в виде строки, массива байт или файла, то соответственно мы можем использовать один из методов `GetStringAsync()/GetByteArrayAsync()/GetStreamAsync()`, либо можно обрабатывать свойство `Content` объекта `HttpResponseMessage`.

Для реализации примера клиента воспользуемся разработанным компанией «DEX» учебным сервером [Swagger UI \(dex-it.ru\)](http://dex-it.ru).

Создадим новый проект по шаблону “Библиотека классов” и добавим в него класс “`TeamService`”, в котором разместим логику работы с сервером.

В первую очередь, нам необходимо авторизоваться для получения доступа к методам апи. Методы `/api/Auth/SignUp` и `/api/Auth/SignIn` допускает анонимное обращение для регистрации нового пользователя и получения токена авторизации соответственно. Токен понадобится для получения доступа к функционалу сервера. Подробнее о том что из себя представляет токен можно почитать тут : [Токен Авторизации / Хабр \(habr.com\)](http://habr.com).

Для упрощения работы рекомендуем пройти регистрацию вручную, заполнив данные и вызвав метод “`SignUp`” прямо в свагере, а алгоритм получения токена мы автоматизируем в примере ниже. Это оправдано, поскольку у токена есть ограничение по “времени жизни”, по истечению которого он становится недействительным и нужно запрашивать новый.

Для хранения созданных в ходе регистрации логина, пароля и ответа от сервера, добавим в проект папку “`Models`” и расположим в ней классы “`AuthData`” и “`AuthResponse`”:

```
public class AuthData
{
    public string Login { get; set; }
    public string Password { get; set; }
}
```

```
public class AuthResponse
{
    public string Name { get; set; }
    public string AvatarUrl { get; set; }
    public string Token { get; set; }
}
```

В классе “TeamService” реализуем метод обеспечивающий получение токена авторизации. Здесь в первую очередь проводим сериализацию данных из объекта AuthData в json формат. Далее создаем экземпляр класса StringContent в параметры которого передаем результата сериализации, тип шифрования и медиатип. Класс StringContent предоставляет содержимое HTTP в виде строки.

Далее вызываем метод PostAsync в параметры которого передаем адрес соответствующего метода на сервере и объект content содержащий наши данные. Ответ сервера приходит в формате json. Десериализуем его в ранее созданный “AuthResponse”.

```

public class TeamService
{
    private string _token;

    public async Task<AuthResponse> Authorize(AuthData data)
    {
        AuthResponse authResponse;

        var serializedData = JsonConvert.SerializeObject(data);
        var content = new StringContent(serializedData, Encoding.UTF8,
"application/json");

        using (var client = new HttpClient())
        {
            HttpResponseMessage responseMessage = await
client.PostAsync("http://dev.trainee.dex-it.ru/api/
Auth/SignIn", content);

            var message = await
responseMessage.Content.ReadAsStringAsync();

            authResponse = JsonConvert
.DeserializeObject<AuthResponse>(message);

        }

        _token = authResponse.Token;

        return authResponse;
    }
}

```

Успешное выполнение метода “Authorize” приведет к тому что в приватном свойстве “_token” сохраниться актуальное значение токена авторизации.

Теперь, аналогичным образом, посредством Post – запроса к методу сервера «/api/Team/Add», реализуем добавление новых данных на сервер. Для чего, добавим в наш класс метод AddTeam(Team team).


```

public async Task Add(Team team)
{
    string serialisedTeam = JsonConvert.SerializeObject(team);

    var content = new StringContent(serialisedTeam,
Encoding.UTF8, "application/json");
    using (var client = new HttpClient())
    {
        client.DefaultRequestHeaders.Authorization =
AuthenticationHeaderValue.Parse(_token);
        await
client.PostAsync("http://dev.trainee.dex-it.ru/api/Team/Add", content);
    }
}

```

```

namespace SampleForHttpClient.Models
{
    public class TeamResponse
    {
        public IEnumerable<Team> Data{ get; set; }
        public int Count { get; set; }
        public int Page { get; set; }
        public int Size { get; set; }
    }
}

```

```

namespace SampleForHttpClient.Models
{
    public class Team
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public DateTime FoudationYer { get; set; }
        public string Division { get; set; }
        public string Conference { get; set; }
        public string ImageUrl { get; set; }
    }
}

```

Отличия в данном случае лишь в том, что перед запросом мы устанавливаем в свойстве “DefaultRequestHeaders.Authorization” класса “HttpClient” полученное значение токена авторизации.

Убедимся что наши данные попали на сервера. Реализуем метод получения списка зарегистрированных игровых команд, обращаясь к методу сервера «/api/Team/GetTeams»

```
public async Task<TeamResponse> GetTeams()
{
    TeamResponse teams;

    using (var client = new HttpClient())
    {
        client.DefaultRequestHeaders.Authorization =
            AuthenticationHeaderValue("Bearer", _token);

        HttpResponseMessage responseMessage = await
            client.GetAsync("http://dev.trainee.dex-it.ru/api/Team/GetTeams");

        responseMessage.EnsureSuccessStatusCode();

        string message = await responseMessage.Content.ReadAsStringAsync();

        teams = JsonConvert.DeserializeObject<TeamResponse>(message);
    }

    return teams;
}
```

Воспользуемся методом `GetAsync` в параметры которого передаем `Uri` вызываемого метода.

После выполнения запроса мы можем получить ответ в виде объекта класса `HttpResponseMessage`.

Для обработки ответа мы можем воспользоваться следующими свойствами данного класса:

- `Content`: полученные данные
- `Headers`: заголовки ответа, полученные от сервера
- `StatusCode`: статусный код ответа

Воспользуемся методом `EnsureSuccessStatusCode()`, для проверки статуса ответа. В случае получения статуса соответствующему коду ошибки, данный метод вызывает исключение.

Если сервер отправляет нам данные в виде `json`, то, получив данные в виде строки с помощью метода `responseContent.ReadAsStringAsync()`, мы можем их десериализовать из `json` с помощью специальных инструментов или библиотек (например `Newtonsoft.Json`).

Данные десериализуются в заранее подготовленную модели TeamResponse и Team, расположите их в ту же папку что и прочие модели.

5. Код Состояния

Код Состояния (StatusCode) – это трехзначный цифровой код, который определяет результат запроса. Например, если клиент запросил при помощи метода GET некий ресурс, и сервер его смог предоставить, такое состояние имеет код 200. Если же на сервере нет запрашиваемого ресурса, он вернет код состояния 404. Есть и много других состояний.

Пояснение – это текстовое отображение кода состояния, для упрощенного понимания человека. Для кода 200 пояснение имеет вид «OK».

В таблице приведены распространенные коды состояния:

Код	Описание
200	Хорошо. Успешный запрос
301	Запрошенный ресурс был окончательно перенесен на новый URI
302	Запрошенный ресурс был временно перенесен на другой URI
400	Неверный запрос - запрос не понят сервером из-за проблем валидации входных данных
401	Несанкционированный доступ — пользователь нет аутентифицирован.
403	Запрещено - у пользователя нет прав для выполнения операций
404	Не найдено - сервер понял запрос, но не нашёл соответствующего ресурса по указанному URI
408	Время ожидания сервером передачи от клиента истекло
500	Внутренняя ошибка сервера—ошибка помешала HTTP-серверу обработать запрос

Практическая часть:

В папке «Services» реализовать сервис «CurrencyService», предоставляющий метод конвертации валют. В рамках этого метода реализовать запросы к серверу предоставляющего функционал конвертации. В качестве сервера можно применить API с бесплатным доступом [Free Currency API | Amdoren](#).

Вопросы для самоконтроля:

- Для чего нужен протокол HTTP?
- Что подразумевается под клиент-серверным взаимодействием в рамках работы данного протокола?
- Что означает ошибка 401?

Ресурсы:

- [Xamarin Forms | Класс HttpClient и отправка запросов \(metanit.com\)](#)
- [Протокол HTTP в Си-Шарп. Классы HttpRequest и HttpResponse \(mycsharp.ru\)](#)
- [HttpResponseMessage.EnsureSuccessStatusCode Метод \(System.Net.Http\) | Microsoft Docs](#)
- [HTTP – Википедия \(wikipedia.org\)](#)

Регулярные выражения (Regular expression)

Регулярные выражения предоставляют мощный, гибкий и эффективный способ обработки текста. Комплексная нотация сопоставления шаблонов регулярных выражений позволяет быстро анализировать большие объемы текста для поиска определенных шаблонов символов, проверять текст на соответствие предопределенному шаблону (например, адресу электронной почты), извлекать, изменять, заменять и удалять текстовые подстроки, а также добавлять извлеченные строки в коллекцию для создания отчета. Для многих приложений, которые работают со строками или анализируют большие блоки текста, регулярные выражения — незаменимый инструмент.

Главный компонент обработки текста с помощью регулярных выражений — это механизм регулярных выражений, представленный в .NET объектом [System.Text.RegularExpressions.Regex](#). Как минимум, для обработки текста с использованием в регулярных выражений механизма регулярных выражений необходимо предоставить два следующих элемента:

1. Шаблон регулярного выражения для определения текста.
2. Текст, который будет проанализирован на соответствие шаблону регулярного выражения.

Методы класса [Regex](#) позволяют выполнять следующие операции:

1. Определить, входит ли шаблон регулярного выражения во входной текст, с помощью метода [Regex.IsMatch](#)

Пример использования `Regex.IsMatch`:

```
class Program
{
    static void Main(string[] args)
    {
        string[] values = { "111-22-3333", "111-2-3333" };
        string pattern = @"^\d{3}-\d{2}-\d{4}$";
        foreach (string value in values)
        {
            if (Regex.IsMatch(value, pattern))
                Console.WriteLine("{0} is a valid SSN.", value);
            else
                Console.WriteLine("{0}: Invalid", value);
        }
    }
}
```

2. Получить один или все экземпляры текста, соответствующего шаблону регулярного выражения с помощью метода [Regex.Match](#) или [Regex.Matches](#).

Пример использования Regex.Matches:

```
class Program
{
    static void Main(string[] args)
    {
        string input = "This is a a farm that that raises dairy cattle.";
        string pattern = @"\\b(\\w+)\\W+(\\1)\\b";
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("Duplicate '{0}' found at position {1}.",
                match.Groups[1].Value, match.Groups[2].Index);
        }
    }
}
```

3. Заменить текст, соответствующий шаблону регулярного выражения, с помощью метода [Regex.Replace](#).

Пример использования Regex.Replace:

```
class Program
{
    static void Main(string[] args)
    {
        string pattern = @"\\b\\d+\\.\\d{2}\\b";
        string replacement = "$$$&";
        string input = "Total Cost: 103.64";
        Console.WriteLine(Regex.Replace(input, pattern,
            replacement));
    }
}
```

Самостоятельно:

- Синтаксис регулярных выражений
- Класс [System.Text.RegularExpressions.Regex](#)
- Как выполнить проверку на соответствие строки заданному формату ([Regex.IsMatch](#))

- Как получить экземпляры текста, соответствующего шаблону регулярного выражения ([Regex.Match](#), [Regex.Matches](#))
- Как заменить текст, соответствующий шаблону регулярного выражения ([Regex.Replace](#))

Ресурсы:

- <https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/regular-expressions>
- <https://docs.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex>
- <https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/regular-expression-language-quick-reference>
- <https://metanit.com/sharp/tutorial/7.4.php>
- <https://regexr.com>

Многопоточность, блокировки, дедлоки

Любое приложение на C# запускается как один поток (Основной поток), но программист может предусмотреть запуск параллельных работ в фоне (Фоновый поток). Различие между фоновым и основным потоком в том, что завершение

основного потока завершает работу приложения и все. Приведем пример.

Пример:

```
class ThreadTest
{
    static void Main()
    {
        var t = new Thread(WriteY);
        t.Start(); // Выполнить WriteY в новом потоке

        while (true)
        {
            Console.Write("x"); // Все время печатать 'x'
        }
    }

    static void WriteY()
    {
        while (true)
        {
            Console.Write("y"); // Все время печатать 'y'
        }
    }
}
```

Запустив вы увидите вывод в консоль букв x и y идущих вперемешку, по мере того как потоки будут делить рабочее время.

Немного теории. Управление многопоточностью осуществляет планировщик потоков, эту функцию CLR обычно делегирует операционной системе. Планировщик потоков гарантирует, что активным потокам выделяется соответствующее время на выполнение, а потоки, ожидающие или заблокированные, к примеру, на ожидании эксклюзивной блокировки, или пользовательского ввода – не потребляют времени CPU.

На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков. Это приводит к непредсказуемому поведению, как в самом первом примере, где каждая последовательность символов 'X' и 'Y' соответствует кванту времени,

выделенному потоку. В Windows типичное значение кванта времени – десятки миллисекунд – выбрано как намного большее, чем затраты CPU на переключение контекста между потоками (несколько микросекунд).

На многопроцессорных компьютерах многопоточность реализована как смесь квантования времени и подлинного параллелизма, когда разные потоки выполняют код на разных CPU. Необходимость квантования времени все равно остается, так как операционная система должна обслуживать как свои собственные потоки, так и потоки других приложений. Говорят, что поток вытесняется, когда его выполнение приостанавливается из-за внешних факторов типа квантования времени. В большинстве случаев поток не может контролировать, когда и где он будет вытеснен.

Какой самый главный вывод можно сделать из этого абзаца? - Поток не может контролировать когда его прервут, а значит при выполнении работ над общими ресурсами поток может быть прерван, сделав работу только наполовину, и оставив данные в несогласованном состоянии, что приводит к потерям данных и сбою программы. Наша задача - научиться работать с разделяемыми данными, и на помощь нам приходят [примитивы синхронизации](#).

Рассмотрим самый часто используемый случай - [синхронизация на уровне областей кода](#), конструкция lock() или Monitor.Enter().

Пример:

```
var dict = new Dictionary<string, string>();
var locker = new object();
var count = 100;
var completed = 0;

ThreadPool.QueueUserWorkItem(_ => // запуск параллельного потока
{
    var c = count;
    while (c-- > 0)
    {
        lock (locker) // синхронизация
        {
            dict.Add("1_" + c, "hello thread1");
        }
    }

    Interlocked.Increment(ref completed); // потокобезопасный инкремент
    числа
});
```

```

ThreadPool.QueueUserWorkItem(_ => // запуск параллельного потока
{
    var c = count;
    while (c-- > 0)
    {
        lock (locker) // синхронизация
        {
            dict.Add("2_" + c, "hello thread2");
        }
    }

    Interlocked.Increment(ref completed); // потокобезопасный инкремент
числа
});

while (completed < 2) // ожидание завершения
{
    Thread.Sleep(25);
}

```

Для того что бы получить синхронизированную область кода для начала создаем объект синхронизации - `locker`, а затем просто пишем конструкцию `lock(locker){ // синхронизированная область }` - все просто.

Синхронизированная область - область в которой может выполняться только один поток до тех пор пока не покинет синхронизированную область. Таким образом даже если runtime прервет выполнение кода в середине синхронизированной области, все равно никакой другой поток не сможет войти внутрь синхронизированной области.

Рассмотри теперь сценарий когда в коде присутствует несколько объектов синхронизации.

Пример:

```

public void DeadlockTest5()
{
    var locker1 = new object();
    var locker2 = new object();

    // thread 1
    ThreadPool.QueueUserWorkItem(_ =>
    {
        lock (locker1) // блокировка 1
        {
            Thread.Sleep(1000);

```

```

        lock (locker2) // блокировка 2
        {
            Console.WriteLine("Thread 1 got both locks");
        }
    });

    // thread 2
    lock (locker2) // блокировка 2
    {
        Thread.Sleep(1000);
        lock (locker1) // блокировка 1
        {
            Console.WriteLine("Thread 2 got both locks");
        }
    }
}

```

Это очень упрощенная схема, в реальной жизни все происходит хитрее и не так наглядно все конечно, но нам для простоты как раз подходит.

Разберем что происходит, 2 потока (thread1, thread2) начинают работать параллельно и когда thread1 запрашивает locker1 а thread2 запрашивает locker2, пока все хорошо, блокировки наложены, далее они ждут по 1 сек и thread1 запрашивает locker2, но locker2 занят thread2 и следовательно thread1 зависнет и будет ждать освобождения блокировки thread2, а thread2, в свою очередь, запросит блокировку locker1 которая захвачена thread1, то и thread2 зависнет и будет ждать пока thread1 не освободит блокировку locker1.

Таким образом наступает взаимная блокировка и потоки из нее сами выйти никогда не смогут (deadlock). Одно из правил которое поможет вам избежать deadlock - блокируйте ресурсы в одинаковой последовательности, т.е. в обоих потоках сначала нужно блокировать locker1 а потом locker2.

Мьютекс

Помимо стандартной синхронизации через lock, есть ещё один, крайне удобный, инструмент управления синхронизацией потоков представляет класс Mutex или мьютекс, который также располагается в пространстве имен System.Threading, для понимания его работы приведем простой пример.

```

int x = 0;
Mutex mutex = new();

//Запустим четыре потока
for (int i = 0; i < 5; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}

void Print()
{
    mutex.WaitOne();//Приостановим текущий поток, чтобы получить наш
//Мьютекс    x = 1;
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
    mutex.ReleaseMutex();    //Освобождаем Мьютекс
}

```

Основную работу по синхронизации выполняют методы `WaitOne()` и `ReleaseMutex()`. Метод `mutex.WaitOne()` приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс `mutex`. Изначально мьютекс свободен, поэтому его получает один из потоков.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода `mutex.ReleaseMutex()`. А мьютекс получает один из ожидающих потоков.

Таким образом, когда выполнение дойдет до вызова `mutex.WaitOne()`, поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

Семафоры являются еще одним инструментом, который предлагает нам платформа .NET для управления синхронизацией. Семафоры позволяют ограничить количество потоков, которые имеют доступ к определенным ресурсам. В .NET семафоры представлены классом `Semaphore`.

Для создания семафора применяется один из конструкторов класса `Semaphore`:

- `Semaphore (int initialCount, int maximumCount)`: параметр `initialCount` задает начальное количество потоков, а `maximumCount` -

максимальное количество потоков, которые имеют доступ к общим ресурсам

- Semaphore (int initialCount, int maximumCount, string? name): в дополнение задает имя семафора
- Semaphore (int initialCount, int maximumCount, string? name, out bool createdNew): последний параметр - createdNew при значении true указывает, что новый семафор был успешно создан. Если этот параметр равен false, то семафор с указанным именем уже существует

Для работы с потоками класс Semaphore имеет два основных метода:

- WaitOne(): ожидает получения свободного места в семафоре
- Release(): освобождает место в семафоре

Практическая часть:

В теле теста, реализовать:

- а) алгоритм добавления клиентов в банковскую систему и одновременный запрос на получение клиентов из базы;
- б) в классе тестов реализовать параллельное начисление денег на счет одного и того же клиента из двух разных потоков;
- в) реализовать одновременное обновление одного и того-же клиента из двух параллельных потоков.

Вопросы для самоконтроля:

- Что такое поток?
- Зачем нужно его использовать?
- Какие преимущества есть у потоков?
- В чём недостатки и опасности использования потоков?
- Что такое DeadLock?
- Как избежать DeadLock?
- Что такое пул потоков?
- Для чего нужен пул потоков?
- Что такое примитивы синхронизации?
- Какие бывают примитивы синхронизации?

- Когда лучше использовать примитивы синхронизации?
- Когда лучше не использовать примитивы синхронизации?
- Что такое Мьютекс?
- Что такое Семафор?
- В чём отличия Семафора от Мьютекса?

Самостоятельно:

- [Потоки](#)
- [Пул потоков](#)
- [Примитивы синхронизации](#)
- [Синхронизированные коллекции](#)

Ресурсы:

- [C# и .NET | Введение в многопоточность. Класс Thread \(metanit.com\)](#)
- [Thread Класс \(System.Threading\) | Microsoft Docs](#)
- [Потоки в C# .NET первые шаги / Хабр \(habr.com\)](#)

Задачи и класс Task

Класс Task описывает отдельную задачу, которая запускается в одном из потоков.

Для определения и запуска задачи можно использовать различные способы.

Первый способ создание объекта Task и вызов у него метода Start:

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));
task.Start();
```

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".

А метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

```
Task task = Task.Factory.StartNew(() =>
    Console.WriteLine("Hello Task!"));
```

В качестве результата метод возвращает запущенную задачу.

Третий способ определения и запуска задач представляет использование статического метода Task.Run():

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

1. Ожидание задачи

Важно понимать, что задачи не выполняются последовательно. Первая запущенная задача может завершить свое выполнение после последней задачи.

Пример:

```
using System;
using System.Threading;

namespace TaskApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task = new Task(Display);
            task.Start();

            Console.WriteLine("Завершение метода Main");

            Console.ReadLine();
        }

        static void Display()
        {
            Console.WriteLine("Начало работы метода Display");

            Console.WriteLine("Завершение работы метода Display");
        }
    }
}
```

Класс Task в качестве параметра принимает метод Display, который соответствует делегату Action. Далее чтобы запустить задачу, вызываем метод Start: task.Start(), и после этого метод Display начнет выполняться во вторичном потоке. В конце метода Main выводит некоторый маркер-строку, что метод Main завершился.

Однако в данном случае консольный вывод может выглядеть следующим образом:

```
Завершение метода Main
Начало работы метода Display
Завершение работы метода Display
```

То есть мы видим, что даже когда основной код в методе Main уже отработал, запущенная ранее задача еще не завершилась.

Чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait:

```
static void Main(string[] args)
{
    Task task = new Task(Display);
    task.Start();
    task.Wait();

    Console.WriteLine("Завершение метода Main");
    Console.ReadLine();
}
```

2. Вложенные задачи

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга.

Например:

```
static void Main(string[] args)
{
    var outer = Task.Factory.StartNew(() =>           // внешняя задача
    {
        Console.WriteLine("Outer task starting...");
        var inner = Task.Factory.StartNew(() =>      // вложенная задача
        {
            Console.WriteLine("Inner task starting...");
            Thread.Sleep(2000);
            Console.WriteLine("Inner task finished.");
        });
    });
    outer.Wait(); // ожидаем выполнения внешней задачи
    Console.WriteLine("End of Main");

    Console.ReadLine();
}
```

Несмотря на то, что здесь мы ожидаем выполнения внешней задачи, но вложенная задача может завершить выполнение даже после завершения метода Main:

То есть в данном случае внешняя и вложенная задачи выполняются независимо друг от друга.

Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение `TaskCreationOptions.AttachedToParent`:

```
static void Main(string[] args)
{
    var outer = Task.Factory.StartNew(() =>          // внешняя задача
    {
        Console.WriteLine("Outer task starting...");
        var inner = Task.Factory.StartNew(() =>      // вложенная задача
        {
            Console.WriteLine("Inner task starting...");
            Thread.Sleep(2000);
            Console.WriteLine("Inner task finished.");
        }, TaskCreationOptions.AttachedToParent);
    });
    outer.Wait(); // ожидаем выполнения внешней задачи
    Console.WriteLine("End of Main");

    Console.ReadLine();
}
```

В данном случае вложенная задача прикреплена к внешней и выполняется как часть внешней задачи. И внешняя задача завершится только когда завершатся все прикрепленные к ней вложенные задачи.

Возвращение результатов из задач.

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты:

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Task<int> task1 = new Task<int>(()=>Factorial(5));
        task1.Start();

        Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

        Task<Book> task2 = new Task<Book>(() =>
        {
            return new Book { Title = "Война и мир", Author = "Л.
Толстой" };
        });
        task2.Start();

        Book b = task2.Result; // ожидаем получение результата
        Console.WriteLine($"Название книги: {b.Title}, автор:
{b.Author}");

        Console.ReadLine();
    }

    static int Factorial(int x)
    {
        int result = 1;

        for (int i = 1; i <= x; i++)
        {
            result *= i;
        }

        return result;
    }
}

```

Во-первых, чтобы задать возвращаемый из задачи тип объекта, мы должны типизировать Task. Например, Task<int> - в данном случае задача будет возвращать объект int.

И, во-вторых, в качестве задачи должен выполняться метод, возвращающий данный тип объекта. Например, в первом случае у нас в качестве задачи выполняется функция Factorial, которая принимает числовой параметр и также на выходе возвращает число.

Возвращаемое число будет храниться в свойстве Result: task1.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

То же самое и со второй задачей task2. В этом случае в лямбда-выражении возвращается объект Book. И также мы его получаем с помощью task2.Result

При этом при обращении к свойству Result программа текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

3. Отмена задач и параллельных операций. CancellationToken

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого .NET предоставляет класс CancellationToken.

Пример:

```
static void Main(string[] args)
{
    CancellationTokenSource cancellationTokenSource = new
    CancellationTokenSource();
    CancellationToken token = cancellationTokenSource.Token;

    int number = 6;

    Task task1 = new Task(() =>
    {
        int result = 1;
        for (int i = 1; i <= number; i++)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Операция прервана");
                return;
            }

            result *= i;
            Thread.Sleep(5000);
        }
        Console.WriteLine($"Факториал числа {number} равен {result}");
    });
```

```

    });
    task1.Start();

    Console.WriteLine("Введите Y для отмены операции или другой символ
для ее продолжения:");
    string s = Console.ReadLine();
    if (s == "Y")
        cancellationTokenSource.Cancel();

    Console.Read();
}

```

Для отмены операции нам надо создать и использовать токен. Вначале создается объект `CancellationTokenSource`:

```

CancellationTokenSource cancellationTokenSource = new
CancellationTokenSource();

```

Затем из него получаем сам токен:

```

CancellationToken token = cancellationTokenSource.Token;

```

Чтобы отменить операцию, необходимо вызвать метод `Cancel()` у объекта `CancellationTokenSource`:

```

cancellationTokenSource.Cancel();

```

В самой операции мы можем отловить выставление токена с помощью условной конструкции:

```

if (token.IsCancellationRequested)
{
    Console.WriteLine("Операция прервана токеном");
    return;
}

```

Если был вызван метод `cancellationTokenSource.Cancel()`, то выражение `token.IsCancellationRequested` возвращает `true`.

Если операция представляет внешний метод, то ему надо передавать в качестве одного из параметров токен.

Практическая часть:

В папке «Services» реализовать сервис «RateUpdater» работающий в фоне и предоставляющий метод ежемесячного начисления процентной ставки каждому клиенту.

Вопросы для самоконтроля:

- Что такое задача и класс Task?
- Зачем нужна задача?
- В чём отличие от потока?
- Как правильно работать с задачей, запускать, ожидать?
- Какие есть состояния у задачи?
- Вложенность задач, как используется и зачем?
- Как отменить задачу?
- Есть ли опасности при использовании CancellationToken?

Самостоятельно:

- Исследовать работу с инструментами класса Parallel (<https://metanit.com/sharp/tutorial/12.4.php>)

Ресурсы:

- <https://highload.today/klass-task-v-c/>
- <https://csharpcooking.github.io/2022/02/27/Task-Scheduler.html>
- <https://dou.ua/lenta/articles/asynchronous-programming>
- <https://habr.com/ru/post/470830/>

Асинхронное программирование. Асинхронные методы, async и await

Асинхронность позволяет вынести отдельные задачи из основного потока в отдельные потоки. Особенно это актуально в графических программах, где продолжительные задачи могут блокировать интерфейс пользователя. И чтобы этого не произошло, нужно задействовать асинхронность. Также асинхронность несет выгоды в веб-приложениях при обработке запросов от пользователей, при обращении к базам данных или сетевым ресурсам. При больших запросах к базе данных асинхронный метод просто уснет на время, пока не получит данные от БД, а основной поток сможет продолжить свою работу. В синхронном же приложении, если бы код получения данных находился в основном потоке, этот поток просто бы блокировался на время получения данных.

Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: `async` и `await`, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

Асинхронный метод обладает следующими признаками:

- В заголовке метода используется модификатор `async`
- Метод содержит одно или несколько выражений `await`
- В качестве возвращаемого типа чаще всего используется один из:
 - `void`
 - `Task`
 - `Task<T>`
 - `ValueTask<T>`

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами `out` и `ref`.

Рассмотрим пример асинхронного метода:

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

```
// вызов асинхронного метода

Console.WriteLine("Введите число: ");
int n = Int32.Parse(Console.ReadLine());
Console.WriteLine($"Квадрат числа равен {n * n}");

    Console.Read();
}

static async Task FactorialAsync()
{
    Console.WriteLine("Начало метода FactorialAsync"); //
выполняется синхронно
    await Task.Run(() => Factorial(6));
// выполняется асинхронно
    Console.WriteLine("Конец метода FactorialAsync");
}

static void Factorial(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }
    Thread.Sleep(3000);
    Console.WriteLine($"Факториал равен {result}");
}
}
```

```
Начало метода FactorialAsync
Введите число:
4
Квадрат числа равен 16
Факториал равен 720
Конец метода FactorialAsync
```

Разберем поэтапно, что здесь происходит:

1. Запускается метод Main, в котором вызывается асинхронный метод FactorialAsync.
2. Метод FactorialAsync начинает выполняться синхронно вплоть до выражения await.
3. Выражение await запускает асинхронную задачу Run(()=>Factorial())

4. Пока выполняется асинхронная задача `Task.Run(()=>Factorial())` (а она может выполняться довольно продолжительное время), выполнение кода возвращается в вызывающий метод - то есть в метод `Main`. В методе `Main` нам будет предложено ввести число для вычисления квадрата числа.

В этом и преимущество асинхронных методов - асинхронная задача, которая может выполняться довольно долгое время, не блокирует метод `Main`, и мы можем продолжать работу с ним, например, вводить и обрабатывать данные.

5. Когда асинхронная задача завершила свое выполнение (в случае выше - подсчитала факториал числа), продолжает работу асинхронный метод `FactorialAsync`, который вызвал асинхронную задачу.

Последовательный и параллельный вызов асинхронных операций

Асинхронный метод может содержать множество выражений `await`. Когда система встречает в блоке кода оператор `await`, то выполнение в асинхронном методе останавливается, пока не завершится асинхронная задача. После завершения задачи управление переходит к следующему оператору `await` и так далее. Это позволяет вызывать асинхронные задачи последовательно в определенном порядке.

Например изменим немного метод `FactorialAsync`:

```
static async Task FactorialAsync()
{
    Console.WriteLine("Начало метода FactorialAsync"); //
    выполняется синхронно
    await Task.Run(() => Factorial(4));
    await Task.Run(() => Factorial(3));
    await Task.Run(() => Factorial(5));                //
    выполняется асинхронно
    Console.WriteLine("Конец метода FactorialAsync");
}
```

Консольный вывод данной программы:

```
Начало метода FactorialAsync  
Введите число:  
4  
Квадрат числа равен 16  
Факториал равен 24  
Факториал равен 6  
Факториал равен 120  
Конец метода FactorialAsync
```

То есть мы видим, что факториалы вычисляются последовательно. И в данном случае вывод строго детерминирован.

Нередко такая последовательность бывает необходима, если одна задача зависит от результатов другой.

Однако не всегда существует подобная зависимость между задачами. В этом случае мы можем запустить все задачи параллельно и через метод `Task.WhenAll` отследить их завершение.

Например, снова изменим метод `FactorialAsync`:

```
static async void FactorialAsync()  
{  
    Task t1 = Task.Run(() => Factorial(4));  
    Task t2 = Task.Run(() => Factorial(3));  
    Task t3 = Task.Run(() => Factorial(5));  
    await Task.WhenAll(new[] { t1, t2, t3 });  
}
```

Вначале запускаются три задачи. Затем `Task.WhenAll` создает новую задачу, которая будет автоматически выполнена после выполнения всех предоставленных задач, то есть задач `t1`, `t2`, `t3`. А с помощью оператора `await` ожидаем ее завершения.

В итоге все три задачи теперь будут запускаться параллельно, однако вызывающий метод `FactorialAsync` благодаря оператору `await` все равно будет ожидать, пока они все не завершатся. И в этом случае вывод программы не детерминирован. Например, он может быть следующим:

```
Факториал числа 5 равен 120  
Факториал числа 4 равен 24  
Факториал числа 3 равен 6
```

Если задача возвращает какое-нибудь значение, то это значение потом можно получить с помощью свойства `Result`.

Практическая часть:

Ограничить пул потоков, в метод получения клиентов по фильтру добавить задержку для имитации долгой работы. В тесте запустить вызов этого метода в цикле, столько раз сколько потоков доступно. Продемонстрировать блокировку системы, попытавшись добавить нового пользователя (сваггер или прямо в тесте).

Переписать методы сервиса с учетом темы асинхронности, повторить эксперимент, оценить разницу, дать объяснение.

Вопросы для самоконтроля:

- Что такое асинхронность?
- В чём отличие от параллельности?
- Зачем нужно использовать асинхронность?
- Всегда ли стоит её использовать?
- Async/Await, как использовать?
- Зачем нужна синхронизация?
- Как использовать синхронизацию?
- Что может возвращать асинхронный метод?
- Как получить данные из асинхронного метода?
- Можно ли получить данные синхронно и сразу?
- Отличия Task от ValueTask?

Ресурсы:

- [Асинхронное программирование на C# | Microsoft Docs](#)
- [Параллелизм против многопоточности против асинхронного программирования: разъяснение / Хабр \(habr.com\)](#)
- [C# и .NET | Асинхронные методы, async и await \(metanit.com\)](#)
- [C# и .NET | Последовательный и параллельный вызов асинхронных операций \(metanit.com\)](#)

Модульное тестирование

До запуска приложения в производство, когда оно станет доступно пользователям, важно убедиться, что данное приложение функционирует, как и должно, что в нем нет ошибок. Для проверки приложения мы можем использовать различные схемы и механизмы тестирования. Одним из таких механизмов являются юнит-тесты.

Юнит-тесты позволяют быстро и автоматически протестировать отдельные компоненты приложения независимо от остальной его части. Не всегда юнит-тесты могут покрыть весь код приложения, но тем не менее они позволяют существенно уменьшить количество ошибок уже на этапе разработки.

Для создания юнит-тестов выбираются небольшие участки кода, которые надо протестировать. Тестируемый участок, как правило, должен быть меньше класса. В большинстве случаев тестируется отдельный метод класса или даже часть функционала метода. Упор на небольшие участки позволяет довольно быстро писать простенькие тесты.

Однажды написанный код нередко читают многократно, поэтому важно писать понятный код. Особенно это важно в юнит-тестах, где в случае неудачи при тестировании разработчик должен быстро прочитать исходный код и понять в чем проблема и как ее исправить. А использование небольших участков кода значительно упрощает подобную работу.

При тестировании важно изолировать тестируемый код от остальной программы, с которой он взаимодействует, чтобы потом четко определить возможность ошибок именно в этом изолированном коде. Что упрощает и повышает контроль над отдельными компонентами программы.

Фреймворки тестирования

Для написания юнит-тестов мы можем сами создавать весь необходимый функционал, использовать какие-то свои способы тестирования, однако, как правило, для этого применяются специальные фреймворки. Некоторые из них:

- **[xUnit.net](#)**: фреймворк тестирования для платформы .NET. Наиболее популярный фреймворк для работы именно с .NET Core и ASP.NET Core
- **MS Test**: фреймворк юнит-тестирования от компании Microsoft, который по умолчанию включен в Visual Studio и который также можно использовать с .NET Core
- **NUnit**: портированный фреймворк с JUnit для платформы .NET

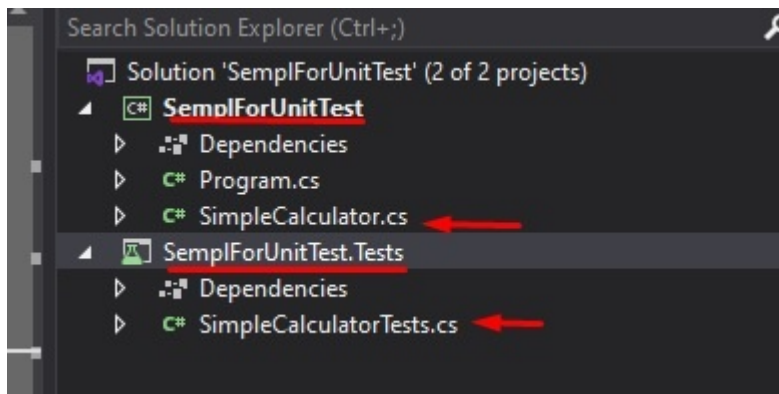
Данные фреймворки предоставляют несложный API, который позволяет быстро написать и автоматически проверить тесты.

Разберем пример опираясь на фреймворк xUnit.

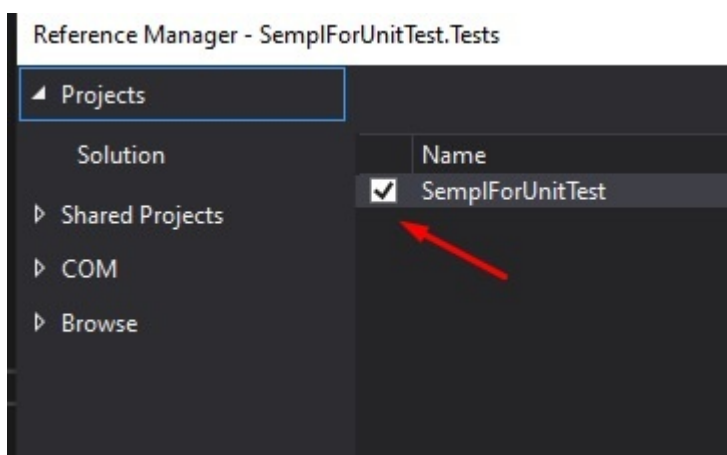
Для начала давайте создадим консольное приложение, и добавим в него класс реализующий простейший функционал калькулятора.

```
public class SimpleCalculator
{
    public int GetSum(int a, int b)
    {
        return a + b;
    }
    public int GetSubstraction(int a, int b)
    {
        return a - b;
    }
    public int GetMultiplication(int a, int b)
    {
        return a * b;
    }
    public int Getdivision(int a, int b)
    {
        return a / b;
    }
}
```

Приведенные методы мы будем подвергать тестированию. Для начала работы над тестами, добавим в это же решение новый проект в виде библиотеки классов. При этом следует придерживаться следующего правила наименования [название тестируемого проекта]+.Tests. В проекта тестов добавим класс который будет содержать сами тесты, имя класса выбираем примерно по тому же принципу [Название тестируемого класса]+Tests. В результате получим такую структуру решения:



Перед началом работы над тестами необходимо в проект тестов добавить ссылку на тестируемый проект, так как нам понадобятся объекты из его пространства имен.



Кроме того, в проект тестов необходимо добавить три NuGet пакета:
Microsoft.NET.Test.Sdk

xunit: функционал фреймворка тестирования

xunit.runner.visualstudio: обеспечивает интеграцию с Visual Studio

Причем вначале лучше добавлять Microsoft.NET.Test.Sdk, так как он содержит ряд зависимостей, используемых в xunit.runner.visualstudio.

Сами тесты представляют собой методы, помеченные специальным атрибутом [Fact] сообщаящий компилятору о том, что это не простой метод.

Название теста следует выбирать по следующему принципу [Имя тестируемого метода]_[сценарий]_[ожидаемый результат]

Например тест для метода «GetSum» следует именовать так «GetSum2Plus5Eq7».

Тело теста как правило условно разделено комментариями на три блока, каждому из которых отведена своя роль:

//Arrange – инициализация тестируемых компонентов

//Act – вызов тестируемого алгоритма, получение результата
//Assert – проверка результата на соответствие ожидаемому значению.

В результате класс тестов принимает следующий вид:

```
using Xunit;
namespace SemplForUnitTest.Tests
{
    public class SimpleCalculatorTests
    {
        [Fact]
        public void GetSum_2_Plus_5_Eq_7()
        {
            //Arrange
            SimpleCalculator calculator = new SimpleCalculator();

            //Act
            int result = calculator.GetSum(2,5);

            //Assert
            Assert.Equal(result, 7);
        }

        [Fact]
        public void GetSubstraction_5_Minus_4_Eq_1()
        {
            //Arrange
            SimpleCalculator calculator = new SimpleCalculator();

            //Act
            int result = calculator.GetSubstraction(5, 4);

            //Assert
            Assert.Equal(result, 1);
        }
    }
}
```

Класс «Assert» предоставляет набор методов позволяющих осуществить разнообразные проверки на соответствие полученного результата ожидаемому.

Запустить тесты можно пользуясь функционалом вкладки «Test», либо нажав правой кнопкой мышки и выбрав пункт «Run Test(s)» из списка.

Удачно выполненные тесты помечаются зеленой галочкой, проваленные красным крестиком.

Практическая часть:

- В папку “Application” добавить проект тестов по шаблону библиотека классов.
- Реализовать практику указанную в теме “Эквивалентность” добавляя соответствующие тесты.

Самостоятельно:

Написать маленький проект простого калькулятора с 4мя основными операциями. Реализацию начать с добавления проекта тестов, с последующей реализацией методов под данные тесты.

Ресурсы:

[ASP.NET Core | Создание юнит-тестов \(metanit.com\)](#)

[Юнит-тестирование для чайников / Хабр \(habr.com\)](#)

[Рекомендации по написанию модульных тестов - .NET | Microsoft Docs](#)