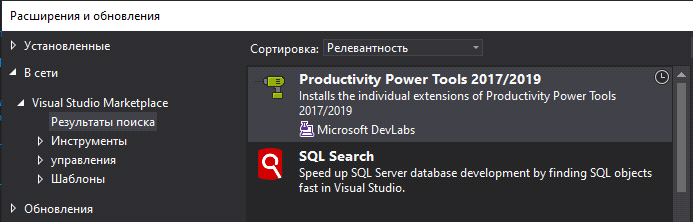
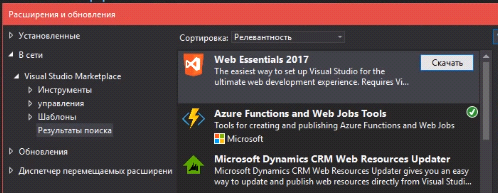
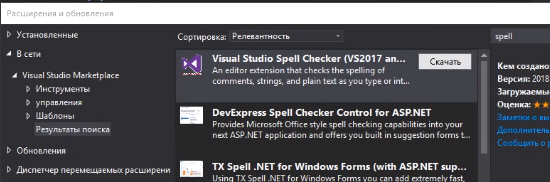
Полезные расширения: 

Для разработки в ВЕБ:



Проверяет грамматику в коде:



- интелисенс – отображает подсказки

**Переменная** – именованная область памяти.

**Stack** (значимые типы) - быстро, но мало

**Heap** (ссылочные типы) - медленно, но много – объем оперативной памяти

**Функциональный стиль** –несколько команд объединяются в именованную функцию, которую можно было вызвать в любом другом участке кода.

**Процедурный стиль** – последовательная запись кода (грубо говоря).

**Принцип черного ящик** – подаем входные данные, не видим внутреннюю часть (то, как работает программа) и получаем готовый результат. Так должны работать классы.

*Пример*: микроволновка. Засунули еду – получили разогретую.

**Принцип белого ящика** – мы вручную настраиваем программу, видим ее «внутренности».

*Пример*: микроволновка. Засунули еду, подрегулировали значения, силу разогрева, кол-во всяких микроволн и прочее, получили результат.

## Парадигмы ООП

**Инкапсуляция** *(сокрытие)* – предоставление только тех инструментов, которыми должен пользоваться пользователь.

**Наследование** – создает зависимость между двумя классами (как бы объединяя их). От базового класса передаем все свойства классу наследнику.

**Полиморфизм** – преобразование класса наследник в базовый класс, откинув все лишние свойства.

**Абстракция/абстрагирование** – учет только тех параметров, которые нужны конкретно в нашей объектной области.

**Посылка и обработка сообщения** – при совершении определенного события, происходит вывод/обработка других событий(-ия).

**Повторение кода** – не изобретаем велосипед, не пишем кода 100500 раз один и тот же, а используем уже существующие библиотеки, функции, методы и тд. (Don’t repeat your self).

//public - публичный открытый - открыт всем и всегда

//internal - открытый в пределах проекта - доступен только в одном проекте

//protected - защищенный - доступный только унаследованному классу от базового, остальным - нет

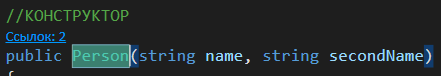
//private - закрытый - принадлежит ТОЛЬКО этому классу

**Экземпляр класса** (англ. instance) — это описание конкретного объекта в памяти. Класс описывает свойства и методы, которые будут доступны у объекта, построенного по описанию, заложенному в классе.

## Методы, свойства, конструкторы

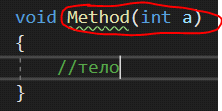
По стандарту все методы, св-ва, конструкторы и другие типы задаются приватными, однако для красоты и эстетики лучше дописать модификатор доступа private.

Все конструкторы создаются от имени класса:



В некоторых ЯП выделяются функции и процедуры. Функции обязательно возвращают нам какое-либо значение. Процедуры же ничего нам не возвращают. В ЯП C# у нас функции и процедуры объединены под одним общим названием - методы. Методы могут как ничего нам не возвращать, так и возвращать значения определенного типа данных.

Сигнатура метода:



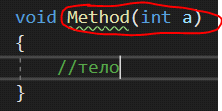
*Правила именования:* гл.+сущ.

В параметры метода можно передавать аргументы, либо же не передавать их совсем.

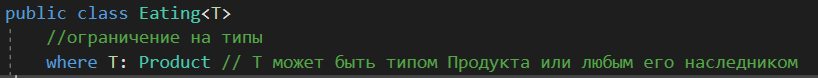
Модификатор доступа static говорит нам о том, что все методы конкретно данного класса могут выполняться без создания объекта от данного класса. Также статические методы не могу работать с нестатическими переменными.

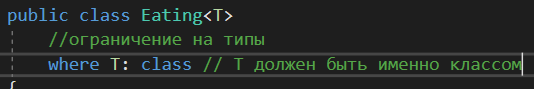
## Перегрузки методов

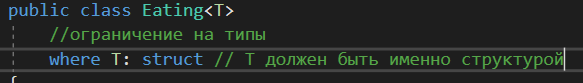
**Сигнатура** - это имя методы и принимаемые типы аргументов и их кол-во.

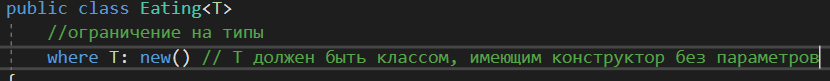


## Generic/Шаблоны/Универсальные типы



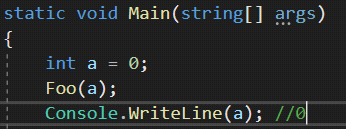


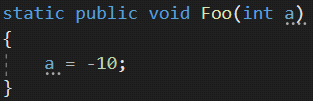




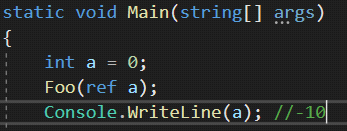
## Ref

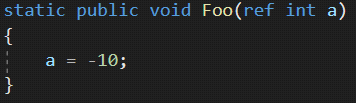
Ключевое слово служит для указания того, что в параметры методы мы будем передавать не ссылку на наши данные, а сами данные (не будем лишний раз их копировать).





В методе мы создали новый элемент (а) и передали ему значения нашей переменной в main (тоже, а). В методе мы присвоили -10 переменной из нашего параметра. После выхода из метода наша «а» в main не изменится.





В данном случае мы указываем, можно сказать, передаем ссылку на сам наш объект (не создавая новый в параметрах => экономим память) и совершаем наши операции с самой переменной («а» из main).



Есть еще *ссылочные локальные* и *возвращаемые ссылочные* значения. Но оно тебе надо, м?

## Строки

Интерполяция строки - замена заполнителей в строке численными значениями.



## NULL

Null – пустая ссылка => значимые типы не могу быть null, поскольку не содержат ссылку, а тупа лежат в стеке.

Null можно использовать только с ссылочными типами.

Автоматически причисляется всем объектам класса (ссылочным типам).

Если же мы инициализируем (например) массив, а после присваиваем ему null, то связь с данными в управляемой куче обрывается, и сборщик мусора очищает данные, чтобы не засорять оперативную память.

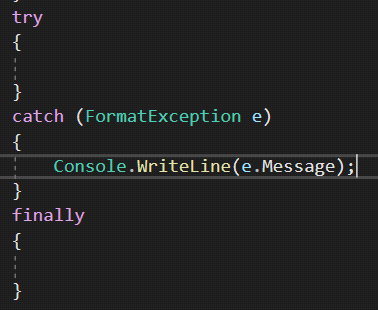
## Interface (доступно на обзор)

*Правило именования*: I (заглавная «ай») + название (прим.: IPerson, IDesktop).

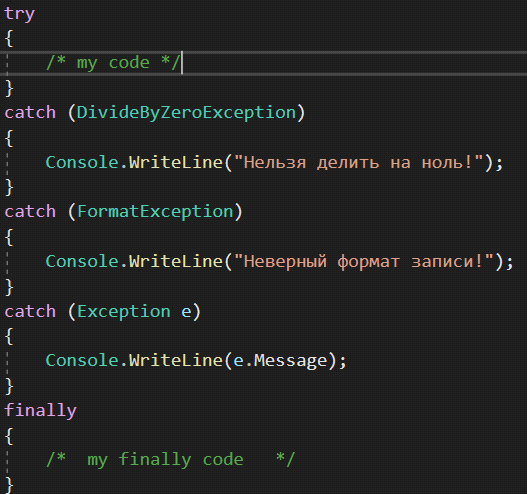
Интерфейсы полностью публичные, не имеют полей и не могут быть приватными. Интерфейсы не наследуются! Интерфейсы *могут реализовываться* или *имплементироваться* (имплементировать – это реализовывать в ПО какой-либо функционал). Интерфейсы желательно использовать, когда задействованы несколько объектных областей (пример: машина, человек, велосипед могу передвигаться => уже можно реализовать для них интерфейс).

## Exception

Базовая конструкция обработки исключений есть: try (в нем помещается код, в котором может возникнуть та или иная ошибка и которую мы должны поймать и обработать), catch («ловить», ловит наше исключение и выполняет код, заключенный в его теле. Также мы можем поместить в нем определенную ошибку, которая может возникнуть), finally (код, который выполняется в любом случае, не зависимо от ошибки. Благодаря ему, мы можем выполнить, например, сохранение какого-либо участка кода, либо же просто данных и корректно попытаться завершить работу приложения).

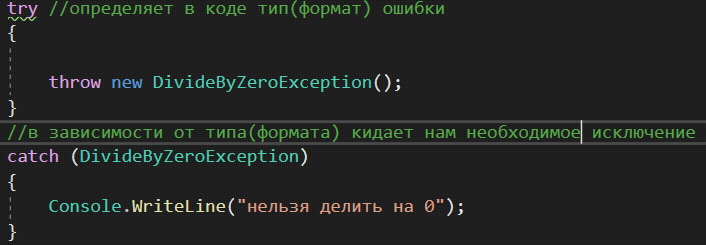


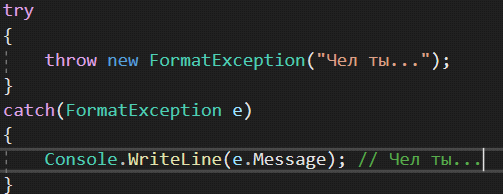
В нашей обработке исключений мы можем использовать несколько блоков catch, чтобы отлавливать различные ошибки в коде, помещенном в try. В зависимости от типа (вида) ошибки, будет выполнен код того catch, который поймал эту ошибку.



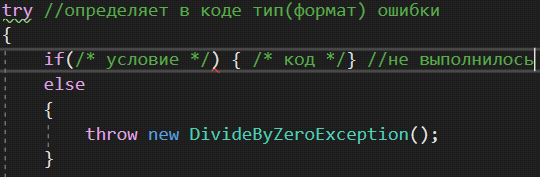
Ошибка Exception ловит и обрабатывает любую ошибку, если предыдущие блоки catch не зафиксировали таковую. «Общий Exception» писать лучше в конце, после записи прочих блоков catch (P.S. чем выше блок catch -> тем обработчик более узконаправленный). Существует множество классов ошибок, который нужно бы подучить и выписать на листочек.

В случае чего, мы можем сами сгенерировать ошибку и доверить наше выполнение кода соответствующему блоку catch

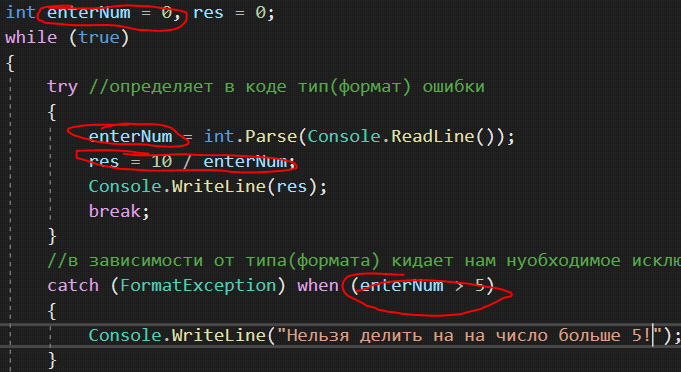




Такое можно проворачивать, если нужно сделать дополнительную проверку условным оператором в блоке try.

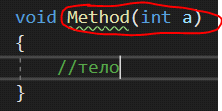
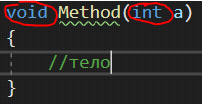


Можно делать фильтры исключений (дополнительное условие выведения ошибки) (when). Тем самым мы можем сделать нашу обработку еще более узконаправленной. Мы можем одну ошибку вызывать из учета дополнительной информации (условия).



Ой, короче… остальное на гите.

## Делегаты

«Это переменная для функций». Сигнатуры – это тип возвращаемого элемента и аргументы метода. Сигнатура метода: . *Сигнатура делегатов*: 

В делегаты мы можем положить любые методы, сигнатуры которых совпадают. Также делегат – это указатель на метод. Являются делегаты коллекциями методов, которые можно последовательно добавлять через “+=” и удалять “-=”.

Если делегат возвращает нам какое-либо значение, то значение будет возвращено последнего добавленного метода.

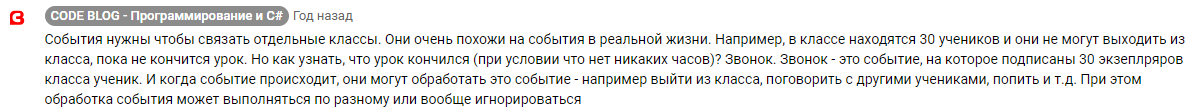
Action – делегат (шаблон делегата), который не возвращает никакого значения. Может принимать в себя от 0 до 16 аргументов. Методы в Action должны быть, как и он сам, void.

Predicate<T> – делегат (шаблон делегата), который возвращает нам bool значение. Принимает в себя минимум один аргумент типа T и может принимать до 16 аргументов.

Func<T> - делегат (шаблон делегата), который возвращает нам последний указанный тип в *параметре* (скобках). Возвращает минимум один аргумент, принимает от нуля до 16.

Event – благодаря ему мы подписываемся на событие.

Подписка на событие происходит после создания делегата.



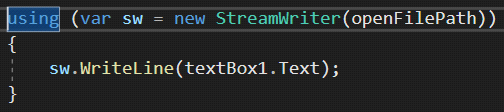
## Потоки (Stream), файлы (file)

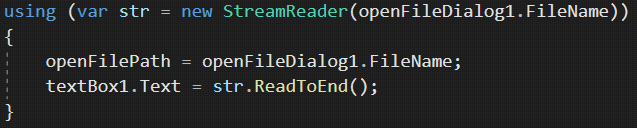
Using – команда, позволяющая создать объект потока и автоматически вызывать команду закрытия при выходе из этой области (проще говоря обеспечение безопасного подключения).

В параметре команды создаем объект потока. Если мы используем using при работе с потоками, то нам не нужно закрывать поток после выполнения работы, нежели бы мы работали в поле Main.

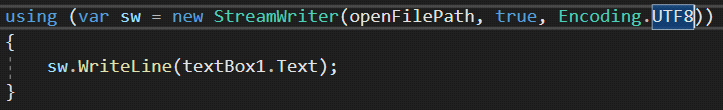
StreamWriter – запись файла. Примечание: если файл не существует, то он тупа создается.

StreamReader – чтение файла.

 - записываем в файл *sw* текст из textbox1.

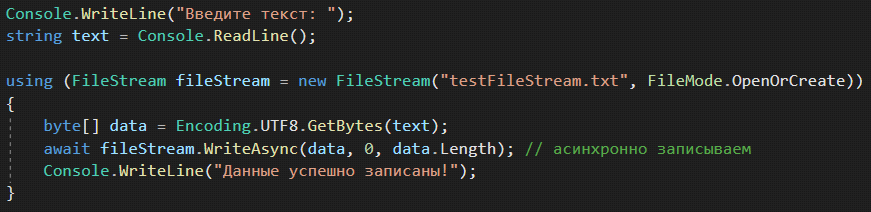
 - просто читаем файл.

Также у этих классов есть конструкторы, в которые они принимают различные параметры. Например, StreamWriter может принимать: <путь к файлу>, <T/F – нужна ли до запись или перезапись, <кодировка>>.

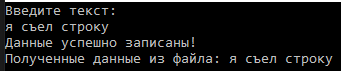


Однако все это работа с исключительно текстовыми файлами, где мы манипулируем самими строками. Но также мы можем записывать и читать бинарные данные. Например:

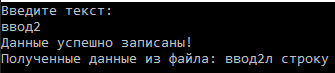




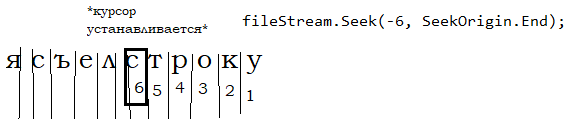
От класса FileStream мы создаем объект для записи данных. В параметры мы передаем path и режим, который хотим использовать, а именно отрыть или создать, если файла не существует. Запись производим сразу асинхронно (потому что можем). В параметр метода передаем сами данные, значение offset (вроде бы, откуда начинать записывать) и длинна самих данных.

Вот что мы получим 

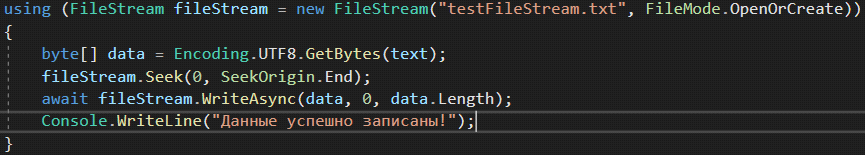
Но если мы таким же способом запишем данные еще раз, но получим следующее:

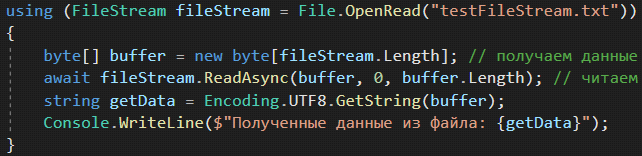


Вторичная запись, как бы на ложилась на предыдущую (т.к. мы записывали все в один документ). В данном случае мы можем воспользоваться методом Seek(), который перемещает наш курсор на заданное кол-во байт.

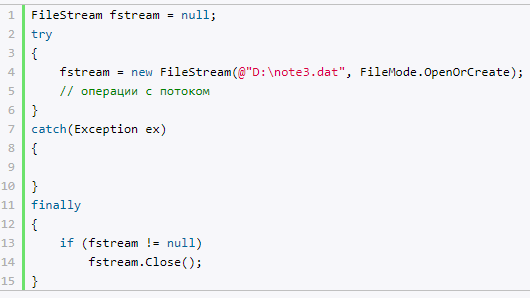


Либо мы можем просто дописать данные, к уже записанным:



Кстати, вот так мы их читаем: 

Все это дело мы писали в конструкции using, которая сама открывала и закрывала поток, однако можно поступить иначе



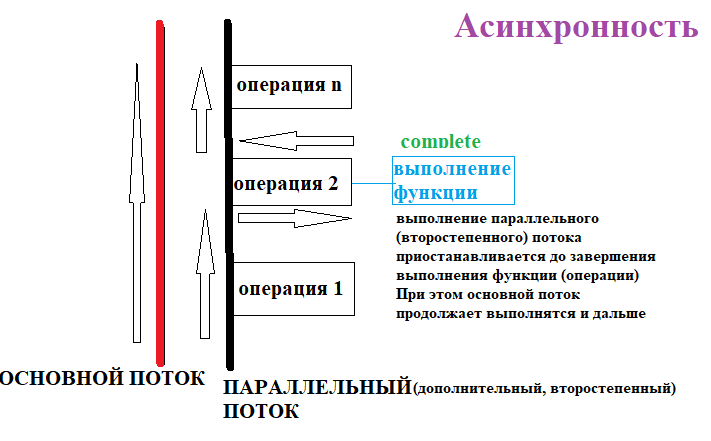
## Асинхронность (async, await) и многопоточность (Thread)

Конкурентность – конкурентность двух или более процессов исполнения за ресурсы машины, при этом они могут никак не взаимодействовать и работать параллельно друг с другом (Пример: конкурентное выполнение двух приложений).

Параллельное выполнение – выделение обособленных частей для параллельного выполнения двух и более процессов.

Многопоточность – взаимодействие двух потоков между собой. Процесс создания и выделения нового потока очень ресурсоемкий

Асинхронность -

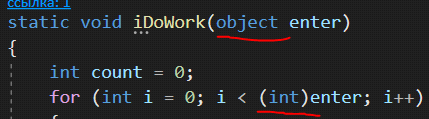


**Многопоточность** создается следующим образом: 

В параметры передаем делегат (ThreadTask, если передаваемый метод без параметров и ParameterizedThreadStart, если у метода есть параметры).



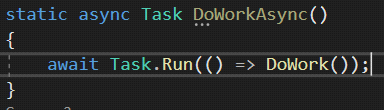
В нашу многопоточную функцию мы передаем параметры с типом object и уже в теле метода производим переопределение.



Вызывается все это дело с помощью метода Start (не забываем передать параметры, если метод у нас принимает что-либо).

**Асинхронный** **метод** создается с указанием класса Task. Чтобы метод был асинхронным, он должен возвращать нам какое-либо значение, либо быть void. Чтобы метод был void, достаточно опустить тип в параметре Task.





Правило именования асинхронного метода:имя + Async.

Ключевое слово async, чтобы показать, что метод асинхронный.

Второе ключевое слово await, чтобы показать, что метод должен выполняться асинхронно. За ним следует вызов метода Run класса Task с лямбда выражением (анонимной конструкцией) и передаем в него наш выполняемый метод.

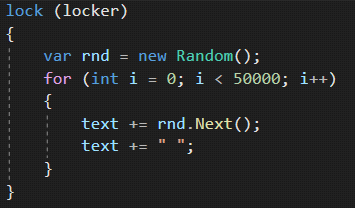
*Немного о работе потоков.*



Наш метод Main синхронный. Код в нем выполняется поочередно. В этом синхронном методе (главном потоке по совместительству) вызываем асинхронный метод 2. Когда мы в методе Main заходим в асинхронный метод и выполняем в нем метод Task.Run, последовательное выполнение кода в Main продолжается, но параллельно с ним выполняется и наш async метод. В методе 2, во время выполнения Task.Run последующий код 3 приостанавливается до завершения выполнения Run.

Чтобы задать приоритетность потоку, нам необходимо приоритетно исполняемый код поместить в локер, изначально создав его глобально (static):





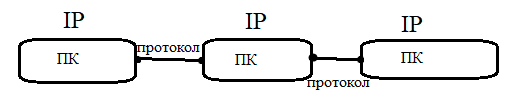
Т.е. у нас есть два потока и у каждого есть метод. Мы помещаем код в каждом методе в lock, тем самым гарантируем, что они выполнятся друг за другом и не произойдет конфликта из-за общих данных (из-за этого, собственно, весь и сумбур).

Больше информации: <https://metanit.com/sharp/tutorial/11.4.php>

Deadlock - ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

## Как устроен интернет

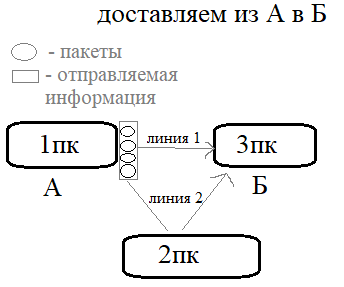
У каждого компьютера есть свой IP. И, казалось бы, зная ip другого компьютера, мы может спокойно посылать на него запросы. Однако, если на линии будут какие-либо технические неполадки, то сигнал просто-напросто потеряется и не дойдет. Для этого используются протоколы.



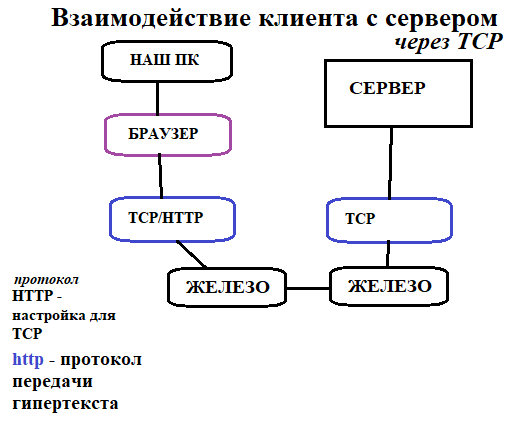
Например, протокол TCP:

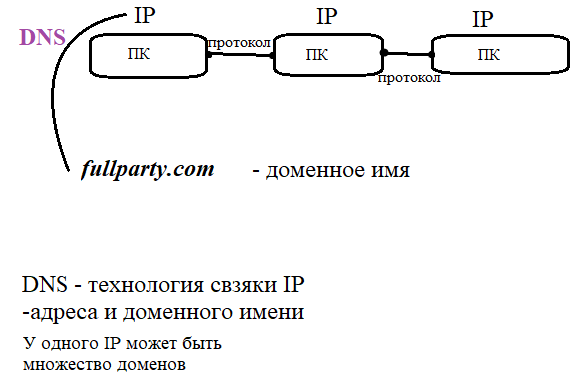
* 100% доставит данные до адресата
* Может отправлять данные потоками
* Разбивает информацию на пакеты
* Выбор быстрого маршрута

Вся отправляемая информация не отправляется цельно – она отправляется пакетами.

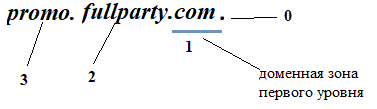
*(отправка пакетами)*

Если на линии 1 будут какие-либо неполадки, то информация отправится другим путем, например, по линии 2.

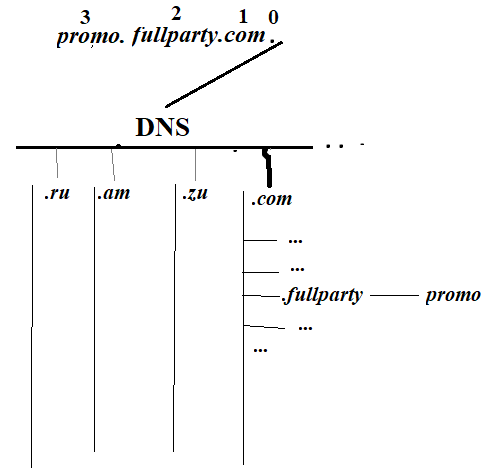
(*работа интернета через TCP*)

*(доменные имена и технология DNS)*

Чтобы серверу не обрабатывать миллиарды компьютеров в поиске нашего доменного имени, у доменов есть доменные зоны.



Как тогда происходит поиск доменного имени? От нулевого доменного уровня поступает запрос в DNS, где он начинает поиск: зона первого уровня –> com => ищет только среди com.

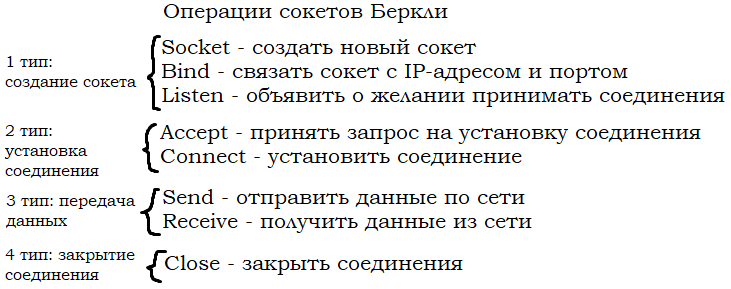
*(поиск по доменным уровням)*

**Про IP**

У IP есть свои форматы. На данный момент используется 4й формат, например, 127.10.0.3 (IPv4) – *32бита*. Есть также шестого формата *(128 бит),* однако на данный момент он не используется, как основной, но все-равно присутствует у каждого компьютера.

**Про сокеты**

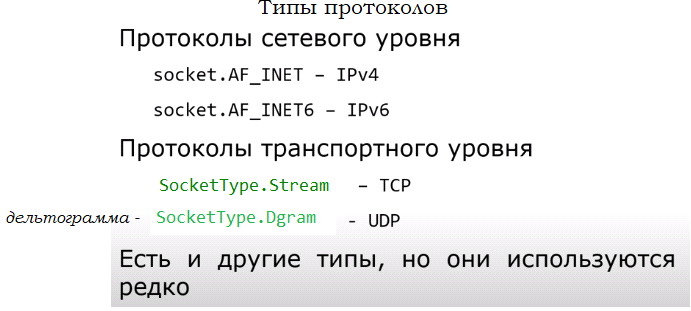
Впервые сокеты использовались в ОС Беркли UNIX. Большинство современных реализаций сокетов построено на операциях Беркли (модель *клиент-сервер*).

*- TCP*

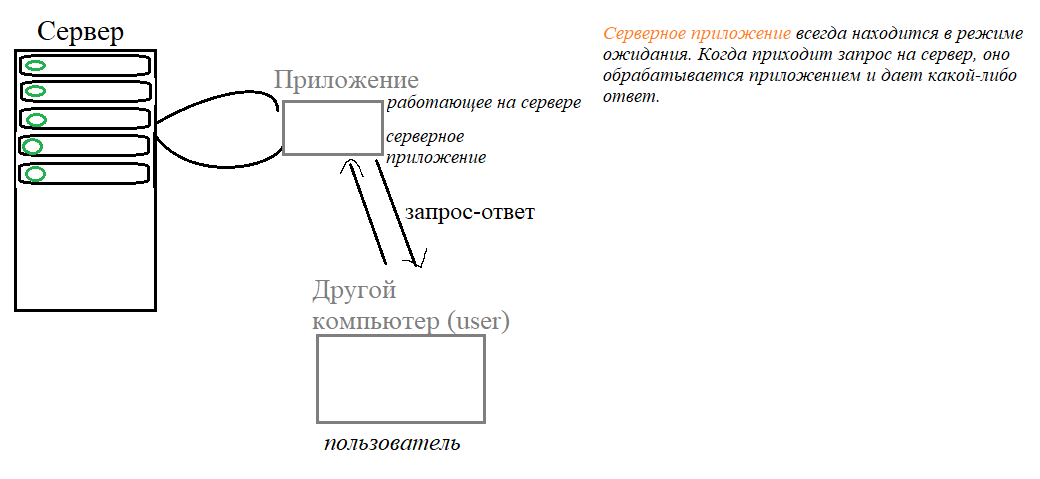
Сервер – работает (слушает) на известном IP-адресе и порту и пассивно ждет запросов на соединение.

Клиент – активно устанавливает соединение с сервером на заданном IP и порту.

После создания сокета на сервере его нужно копировать, чтобы пользователи подключались не к исходному сокету, а к «рабочему». Тем самым сокетов может быть много => может прослушиваться большая аудитория.



## Сокеты (socket) и клиент-серверное взаимодействие по протоколам TCP и UDP



Протокол **TCP** – устанавливает более надежное соединение между двумя точками, гарантирует доставку данных, однако работает медленнее из-за медленного, безопасного соединения.

Протокол **UDP** – работает быстро, не гарантирует доставку всех данных, но подключается быстрее. Так же сервер быстрее обрабатывает запросы от пользователей и может одновременно обрабатывать большее кол-во запросов, чем TCP. Может обеспечивать широковещательную реализацию (сразу многим).

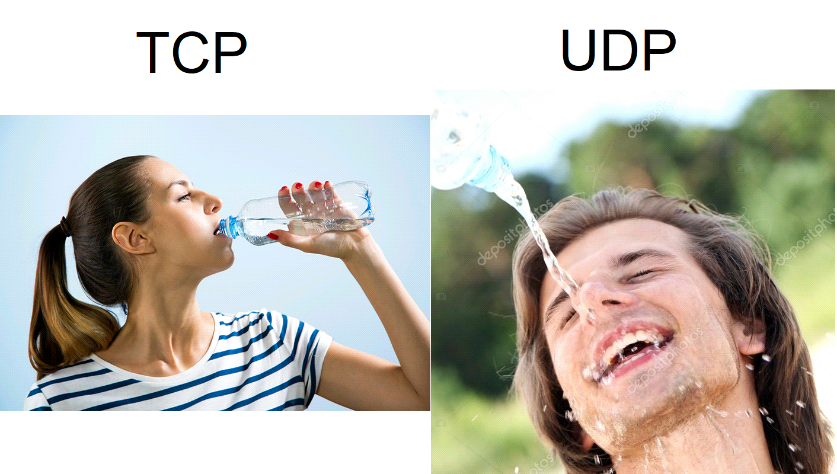
Чтобы установить взаимодействие, над нужно установить конкретный адрес в сети: IP, Port.

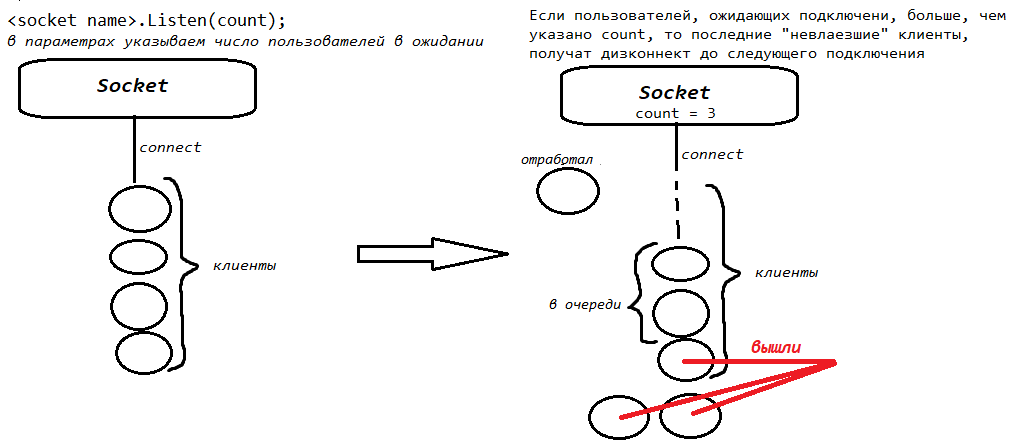
Чтобы установить локальный хост (взаимодействие между приложениями на своей машине) можно воспользоваться локальным хостом: 127.0.0.1 (IP).

const string ip = "127.0.0.1";

End Point – точка подключения.

Socket – это то, через что устанавливается соединение с end point.

(TCP UDP meme)



ПРИ СОЗДАНИИ КЛИНЕТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ НЕОБХОДИМО УЧИТЫВАТЬ, ЧТО СЕРВЕР ДОЛЖЕН РАБОТАТЬ МНОГОПОТОЧНО, ТК РАБОТА ОДНОГО СОКЕТА = ОДИН ПРОЦЕСС.

## LINQ

Linq – язык запроса данных.

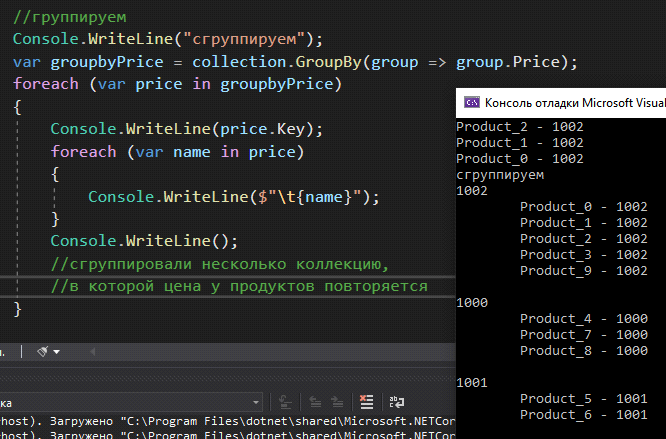
Некоторые ключевые слова: from – куда; in - в какой коллекции; where – где (условие); select – выбрать; orderby – сортировать по (свойство).

Методы расширения:

OrderBy() – сортировать по (свойство) – от минимального к максимальному.

OrderByDescending() - сортировать по (свойство) – от максимального к минимальному.

GroupBy() – сгруппировать по (свойство) – от мин к макс.

*GroupBy*

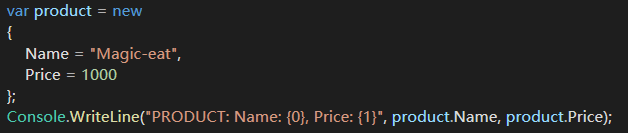
All() – ищет совпадения (условие) – возвращает true, если ВСЕ элементы ему соответствуют.

Any() – ищет совпадения (условие) – возвращает true, если хотя бы один элемент ему соответствуют.

Contains() – ищет совпадения в коллекции по заданному объекту (объект) – если объект является частью коллекции, то возвращает true

## Anonimous types (анонимные типы)

Очередной синтаксический сахар, который позваляет создавать объект со свойствами от не существующего класса. Доступен созданный объект исключительно в метода, в котором был создан. Не может передаваться в конструкторы и как либо перемещаться по проекту.



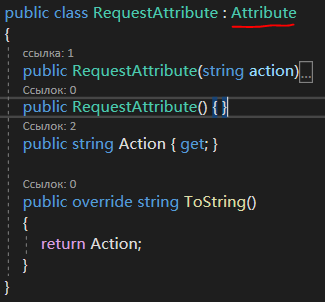
## Кортежы (Typle)

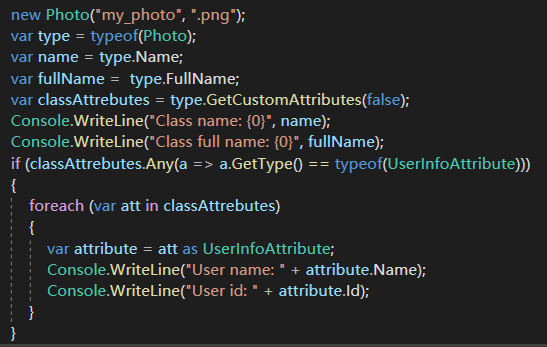
Не изучал че это такое, но на протяжении всего стрима code blog слышал лишь "По возможности старайтесь это не использовать"

## Attribute (Атрибуты) и рефлексия

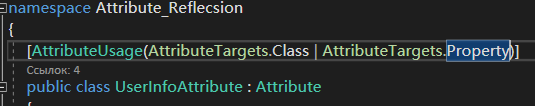
Атрибуты, если кратко, это указания к данным (т.е. Мета-данные). Атрибуты можно применять практически ко всем полям класса (поля, методы, структуры, даже классы, сво-ва и другие). Для того, чтобы создать свой атрибут, необходимо создать класс, наследуемый от System.Attribute

Правило именования: attribute\_name + Attribute





Так же мы можем добавлять ограничения по использование нашего атрибута:



Используя на нашем атрибуте атрибут (AttributeUsage), мы можем указать всех, кто может использовать наш атрибут (класс, свойства, методы и т.д.)

## Entity Framework

Фреймворк для работы с БД SQL.

Особенности:

* Делает кэшированную копию запросов в памяти ПК, прежде чем отправлять запрос напрямую в БД

В консоли диспетчера пакетов можно ввести команды(Вид→другие окна→консоль диспетчера пакетов)

*enable-migrations* – то-есть разрешить делать миграции в БД (добавлять новый столбцы в БД)

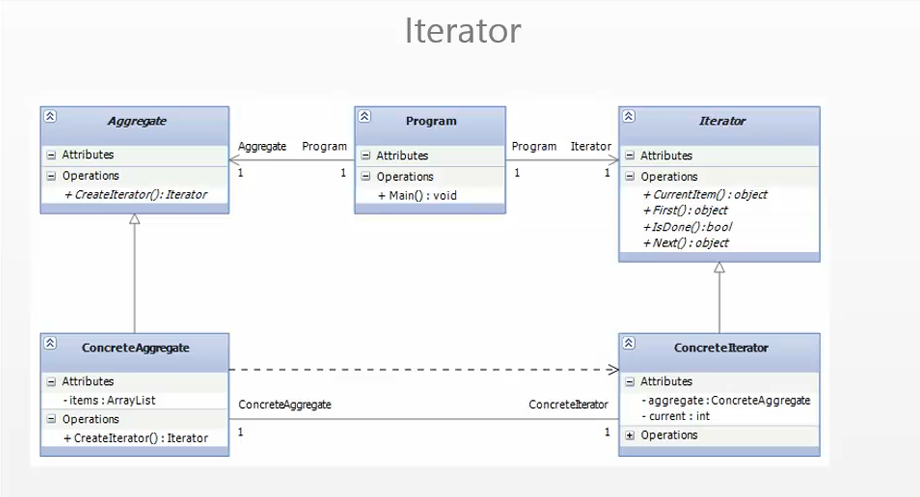
​*add-migration* Prop-name – создать новый столбец (свойство)

​*update-database* – применить изменения

## Пользовательские коллекции и паттерн Iterator

Iterator – паттерн, идея которого заключается в том, чтобы предоставлять последовательный доступ ко всем элементам коллекции, не раскрывая его внутреннего устройства.

За возможность последовательного предоставления данных из коллекции должна отвечать не сама коллекция, а отдельный класс Iterator.



Задача абстрактного класса Aggregate реализовать конкретную коллекцию, а также создать итератор для этой коллекции.

Сущности Iterator:

Current : object – возвращает объект коллекции, на котором установлен указатель

First : object – возвращает первый объект коллекции

IsDone() : bool – метод-предикат, возвращающий true, если указатель достиг последнего элемента коллекции

Next() : вернуть элемент и переместить указать

Isolated storage (изолированное хранилище)

Изолированное хранилище – это возможность сохранять на локальный компьютер (жесткий диск) данные, не получая при этом разрешения на предоставление прав администратора (и прочего) - (короче тупа сохраняем и не паримся емае)

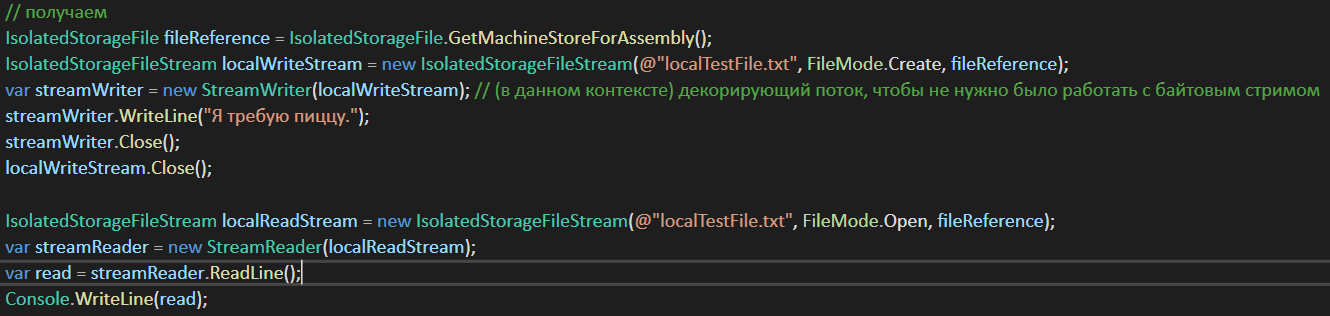
Есть два уровня разграничения доступа к изолированному хранилищу:

1. Это тупа ссылка на локальную машину. Каждая машина – индивидуальное хранилище данных. 2. Это тупа для всех пользователей на локалке



Короче говоря, шобы все это дело хорошо знать надо учиться, объясняю, - мне лень. Короче пишем этот кукиш сверху и кайфуем ептер.

И еще немного псевдокода.



Отличие SDK от API

**API** (Application programming interface) - набор инструкций (сторонняя библиотека / .dll) предназначенная для взаимодействия со сторонними ресурсами (API VK, Google API, Firebase API)

**SDK** (Software development kit) - это набор инструментов для работы с чем-то.

Вы можете рассматривать свой любимый язык программирования как API (набор правил синтаксиса и поведения), а свою любимую среду - как SDK (набор инструментов для написания, проверки и отладки кода).

**SDK** - это набор функционала(библиотек) и утилит для разработки. Собственно, **SDK** и предоставляет реализацию некоторого **API**, это оболочка **API's**, которая упрощает работу для разработчиков.