Плывите сосиски (сосите плывиски))))

[**===1. Системы контроля версий. Классификация. Git, основные возможности**](#_7amqgnhnumnw) **5**

[**===2. Состав. NET Framework. Структура среды выполнения CLR.**](#_274wg0q08mu4) **7**

[**===3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.**](#_7tvslk3ztz37) **8**

[**===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.**](#_xgzgdd6jsabh) **9**

[**===5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение**](#_3882iqo8upx) **11**

[**===6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.**](#_h6bohmrzr536) **13**

[**===7. Неявная типизация – назначение и использование.**](#_8sgya1fc66n) **16**

[**===8. Массивы C# одномерные, прямоугольные и ступенчатые.**](#_bnh17vquy2nd) **17**

[**===9. Понятие кортежей. Свойства, создание**](#_rweq1rux0lyi) **18**

[**===10. Принципы объектно-ориентированного программирования.**](#_7rzj8rakmwfi) **19**

[**===11. Класс. Элементы класса. Свойства и индексаторы.**](#_otg05e818ij8) **20**

[**===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.**](#_uni0yd1f0j2) **22**

[**===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.**](#_5kxncni0hefs) **22**

[**===14. Класс. Конструкторы и их свойства. Деструкторы**](#_ywnpeel3ri9s) **24**

[**===15. Класс и методы System.Object.**](#_57bzabi2jrq) **25**

[**===16. Статические методы и статические конструкторы класса.**](#_71yb8lmg42hk) **26**

[**===17. Статические классы. Методы расширения и правила их определения.**](#_qv9nofwk0t69) **27**

[**===18. Анонимные типы.**](#_60qkucfr2yl5) **28**

[**===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.**](#_1u69n72sibuu) **29**

[**===20. Перегрузка методов и операторов. Правила перегрузки операторов.**](#_8miqlfn98c30) **31**

[**===21. Операции преобразования типа. Явная и неявная форма. Ограничения.**](#_9aobmmwrtqvk) **32**

[**===22. Вложенные типы. Вложенные объекты**](#_m0pq2d1s4obw) **34**

[**===23. Правила наследования C#.**](#_frerxjds3t6b) **35**

[**===24. Сокрытие имен при наследовании. Обращение к скрытым членам**](#_laeumnhr3wop) **36**

[**===25. Использование операций is и as**](#_h88s66qicypi) **37**

[**===26. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.**](#_2sa4szet7kui) **38**

[**===27. Понятие раннего и позднего связывания.**](#_kdxxum623v4e) **39**

[**===28. Абстрактные классы и методы. Бесплодные классы.**](#_atk643nh56jv) **40**

[**===29. Структуры в C#.**](#_4b1h4s712ypy) **41**

[**===30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.**](#_d5ybnbgh036x) **43**

[**===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.**](#_thhomwlifctz) **44**

[**===32. Ковариантность интерфейсов. Контравариантность интерфейсов**](#_j96ls140y8wt) **46**

[**===33. Стандартные интерфейсы .NET. Назначение и применение.**](#_t48awy2ddmxv) **48**

[**===34. Исключительные ситуации. Генерация и повторная генерация исключений.**](#_ewec33w6eul4) **50**

[**===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений**](#_ndn4ndy97317) **51**

[**===36. Обобщения (generics). Свойства обобщений.**](#_gfngvgxe1tr1) **53**

[**===37. Концепция ограничений обобщений. Статические члены обобщений.**](#_w7qbmyvasi5y) **55**

[**===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.**](#_7zkw8kq8i699) **58**

[**===39. Анонимные функции. Лямбда-выражения.**](#_jr1ez2j2k1ns) **60**

[**===40. Обобщённые делегаты .NET. Action, Func, Predicate**](#_ckt8hlvf6d4) **62**

[**===41. События и делегаты.**](#_f1wf07wd8vjp) **63**

[**===42. Стандартные коллекции .NET. Типы коллекций.**](#_1u6hl884n2tp) **65**

[**===43. Стандартные интерфейсы коллекций.**](#_3tucywn9cyni) **67**

[**===44. IEnumerable и IEnumerator**](#_fi8us788c9bv) **69**

[**===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.**](#_r63o2b7f2pm8) **69**

[**===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy**](#_ubdve0sx3jgf) **73**

[**===47. Рефлексия. System Type.**](#_41ecczxgjq35) **77**

[**===48. Классы для работы с файловой системой.**](#_lhpsvmqqyxh8) **78**

[**===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.**](#_3zh4cicqfn0l) **80**

[**===50. Классы адаптеры потоков.**](#_4qy9nlw3s8wu) **82**

[**===51. Сериализация. Форматы сериализации.**](#_bl7a1epyecag) **85**

[**===52. Сериализация контрактов данных. интерфейс ISerializable.**](#_kmh5rs4e93o8) **88**

[**===53. Атрибуты. Создание собственного атрибута.**](#_8zkew264v0mm) **91**

[**===54. Процесс. Домен приложений. Поток выполнения.**](#_949jjdrjbsql) **93**

[**===55. Создание потоков , классы приоритетов. Состояния потоков**](#_hyvcaqg54ntl) **96**

[**===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore**](#_ampxa5g702ix) **99**

[**===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.**](#_oluv7xv50tyg) **104**

[**===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.**](#_xttpk6gc4db0) **106**

[**===59. Параллелизм при императивной обработке данных. Класс Parallel**](#_cis53sjzzzlj) **108**

[**===60. Асинхронные методы. async и await**](#_h9j54ilahosy) **111**

[**===61. Проектирование отношений. Агрегация, композиция и ассоциация**](#_ayhzpsv76cru) **112**

Вопросы

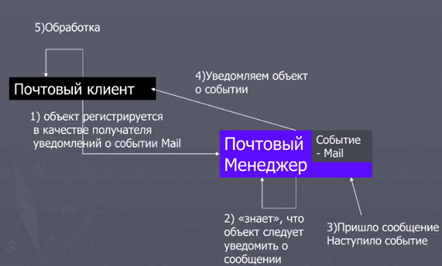
# ===41. События и делегаты.

**Делегаты** - объекты предназначенные для хранения ссылок на методы. Используются для поддержки событий.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

**События** - элемент класса , позволяющий ему посылать другим объектам уведомления об изменении своего состояния.

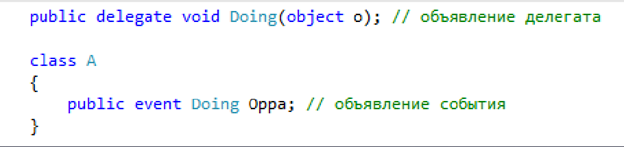
Модель “публикация – подписка” или паттерн “наблюдатель” , класс , являющийся отправителем , сообщения , публикует события, которые он может инициировать , а другие классы , являющиеся получателями сообщения , подписываются на получение этих событий.



События объявляются в классе с помощью ключевого слова **event**, после которого идет название делегата.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

* Описание делегата, задающего сигнатуру обработчиков событий
* Описание события
* Описание метода (методов), инициирующих событие



1. Обработка событий выполняется в классах-получателях
2. сигнатура методов-обработчиков событий, == типу делегата
3. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод :+= и -=
4. Поддерживается групповая адресация

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.

**Стандартные .NET делегаты**

Правила:

1. имя делегата заканчивается суффиксом EventHandler;
2. делегат получает два параметра: первый параметр задает источник события и имеет тип object; второй параметр задает аргументы события и имеет тип EventArgs или производный от него.
3. Имя обработчика события принято составлять из префикса On и имени события;
4. Классы содержащие информацию о событиях должны наследовать от типа System.EventArgs, а имя типа должно заканчиваться словом EventArgs

+делегаты

Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать действия. Анонимный делегат (без создания классанаблюдателя): s.Go += new SomeDelegat(()=> { Console.WriteLine("Я слышу что идете!"); });

Делегаты и события обеспечивают взаимодействие взаимосвязанных объектов. События включены во многие стандартные классы .NET, используемые для разработки Windows-приложений.

# ===42. Стандартные коллекции .NET. Типы коллекций.

В C# **коллекция** представляет собой совокупность объектов. Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

**Типы коллекций:**

* необобщенные (наличие разнотипных данных, ссылки на данные типа object (не обеспечивают типовую безопасность), System.Collections)
* Обобщенные (обеспечивают типовую безопасность, System.Collections.Generic)
* специальные (System.Collections.Specialized)
* с поразрядной организацией (BitArray)
* Параллельные (многопоточный доступ к коллекции, System.Collections.Concurrent)

**Необобщенные коллекции** (ArrayList (IList, ICollection, IEnumerable, ICloneabl), BitArray (ICollection, IEnumerable, ICloneable), Hashtable, Queue, SortedList (класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу), Stack).

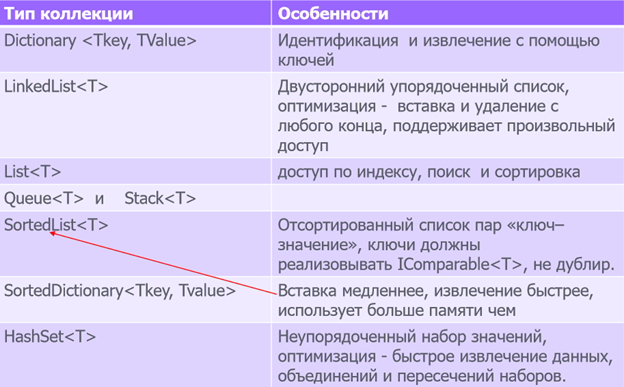
Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных.

Хранят ссылки на объекты, при сохранении или извлечении элементов требуется приведение типов (исключение BitArray). Включены в библиотеку с целью обратной совместимости с существующими приложениями применять не рекомендуется. B UWP эти классы недоступны.

**Специальные коллекции.**

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

**Обобщенные коллекции** (Dictionary <Tkey, TValue>, LinkedList<T>, List<T>, Queue<T>, SortedDictionary<Tkey, Tvalue>, SortedList<T> (использовании памяти и в скорости вставки и удаления), HashSet<T> и SortedSet<T>, Stack<T>).



Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

**Параллельные коллекции.**

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен System.Collections.Concurrent.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций. Коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений

**Поразрядная коллекция.**

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это BitArray. Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

# ===43. Стандартные интерфейсы коллекций.

В C# коллекция представляет собой совокупность объектов.

Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

**IEnumerable<T>** — интерфейс IEnumerable необходим, когда с коллекцией используется оператор foreach. Этот интерфейс определяет метод GetEnumerator(), возвращающий перечислитель, который реализует IEnumerator.

**ICollection<T>** — это интерфейс, реализованный классами обобщенных коллекций. С его помощью можно получить количество элементов в коллекции (свойство Count) и скопировать коллекцию в массив (метод СоруТо()). Можно также добавлять и удалять элементы из коллекции (Add(), Remove(), Clear()).

**IList<T>** интерфейс IList<T> предназначен для создания списков, элементы которых доступны по своим позициям. Этот интерфейс определяет индексатор, а также способы вставки и удаления элементов в определенные позиции (методы Insert() и Remove()). IList<T> унаследован от ICollection<T>.

**ISet<T>** интерфейс ISet<T> появился в версии .NET 4. Этот интерфейс реализуется множествами. Он позволяет комбинировать различные множества в объединения, а также проверять, не пересекаются ли два множества. ISet<T> унаследован от ICollection<T>.

**IDictionary<TKey, TValue>** интерфейс IDictionary<TKey, TValue> реализуется обобщенными классами коллекций, элементы которых состоят из ключа и значения. С помощью этого интерфейса можно получать доступ ко всем ключам и значениям, извлекать элементы по индексатору типа ключа, а также добавлять и удалять элементы.

**ILookup<TKey, TValue>** подобно IDictionary<TKey, TValue> поддерживает ключи и значения. Однако в этом случае коллекция может содержать множественные значения для одного ключа.

**IComparer<T> интерфейс IComparer<T>** реализован компаратором и используется для сортировки элементов внутри коллекции с помощью метода Compare().

**IEqualityComparer<T>** интерфейс IEqualityComparer<T> реализован компаратором, который может быть применен к ключам словаря. Через этот интерфейс объекты могут быть проверены на предмет эквивалентности друг другу. В .NET 4 этот интерфейс также реализован массивами и кортежами.

**IProducerConsumerCollection<T>** интерфейс IProducerConsumerCollection<T> был добавлен в версию .NET 4 для поддержки новых, безопасных в отношении потоков классов коллекций.

# ===44. IEnumerable и IEnumerator

К элементам коллекции нередко приходится обращаться циклически, например, для отображения каждого элемента коллекции. С этой целью можно, с одной стороны, организовать цикл foreach, а с другой — воспользоваться перечислителем.

**Перечислитель** — это объект, который реализует необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T>.

Интерфейс **IEnumerator** представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле foreach. А интерфейс **IEnumerable** через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable(IEnumerable<T>) является базовым для всех коллекций.

**Интерфейс IEnumerator** содержит следующие методы и свойства:

**Метод** **MoveNext()** перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

**Свойство Current** возвращает объект в последовательности, на который указывает указатель.

**Метод Reset()** сбрасывает указатель позиции в начальное положение.

# ===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.

**LINQ** (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.LINQ.

Простейшее определение запроса LINQ выглядит следующим образом:

*from переменная in набор\_объектов*

*select переменная;*

**Грамматика выражений запросов:**

1. Начало - from
2. 0..\* from, let или where.
3. orderby, ascending или descending
4. select или group.
5. конструкции into, join, или повторение с п.2.

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

class LINQQueryExpressions

{

static void Main()

{

// Specify the data source.

int[] scores = new int[] { 97, 92, 81, 60 };

// Define the query expression.

IEnumerable<int> scoreQuery =

from score in scores

where score > 80

select score;

// Execute the query.

foreach (int i in scoreQuery)

{

Console.Write(i + " ");

}

}

}

// Output: 97 92 81

Есть два способа выполнения запроса LINQ: отложенное и немедленное выполнение.

**Отложенное выполнение**

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке. Рассмотрим отложенное выполнение:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// выполнение LINQ-запроса

foreach (string s in selectedTeams)

Console.WriteLine(s);

То есть фактическое выполнение запроса происходит не в строке определения: *var selectedTeams = from t...*, а при переборе в цикле *foreach*.

Фактически LINQ-запрос разбивается на три этапа:

Получение источника данных

Создание запроса

Выполнение запроса и получение его результатов

После определения запроса он может выполняться множество раз. И до выполнения запроса источник данных может изменяться.

Теперь выборка будет содержать два элемента, а не три, так как второй элемент после изменения не будет соответствовать условию.

Важно понимать, что переменная запроса сама по себе не выполняет никаких действий и не возвращает никаких данных. Она только хранит набор команд, которые необходимы для получения результатов. То есть выполнение запроса после его создания откладывается. Само получение результатов производится при переборе в цикле foreach.

**Немедленное выполнение запроса**

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент. Например, Count(), Average(), First() / FirstOrDefault(), Min(), Max() и т.д. Например, метод Count() возвращает числовое значение, которое представляет количество элементов в полученной последовательности. А метод First() возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройтись по ней циклом foreach, который вызывается неявно внутри структуры запроса.

Рассмотрим пример с методом Count(), который возвращает число элементов последовательности:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

// определение и выполнение LINQ-запроса

int i = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t select t).Count();

Console.WriteLine(i); //3

teams[1] = "Ювентус";

Console.WriteLine(i); //3

Результатом метода Count будет объект int, поэтому сработает немедленное выполнение.

Сначала создается запрос: from t in teams where t.ToUpper().StartsWith("Б") orderby t select t. Далее к нему применяется метод Count(), который выполняет запрос, неявно выполняет перебор по последовательности элементов, генерируемой этим запросом, и возвращает число элементов в этой последовательности.

Также для немедленного выполнения LINQ-запроса и кэширования его результатов мы можем применять методы преобразования ToArray<T>(), ToList<T>(), ToDictionary() и т.д.. Эти методы получают результат запроса в виде объектов Array, List и Dictionary соответственно.

# ===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy

**Операция Where.**

Операция Where используется для фильтрации элементов в последовательность. Операция Where имеет два прототипа.

**Первый прототип Where.**

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

Этот прототип Where принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает true.

Поскольку это расширяющий метод, входная последовательность в действительности не передается, поскольку операция Where вызывается с использованием синтаксиса метода экземпляра.

При вызове Where передается делегат метода-предиката. Этот метод-предикат должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать bool.

Операция Where вызовет метод-предикат для каждого элемента входной последовательности и передаст ему этот элемент. Если метод-предикат вернет true, то Where выдаст этот элемент в выходную последовательность Where. Если метод-предикат вернет false, то Where этого не сделает.

**Второй прототип Where.**

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, int, bool> predicate);

Второй прототип Where идентичен первому, но с тем отличием, что он указывает на то, что делегат метода-предиката принимает дополнительный целочисленный аргумент. Этот аргумент будет индексом элемента во входной последовательности.

Нумерация индекса начинается с нуля, поэтому индексом первого элемента будет 0. Последний элемент имеет номер, соответствующий количеству элементов в последовательности минус 1.

**Select.**

Операция Select используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа. Эти типы не обязательно должны совпадать. Существуют два прототипа этой операции.

**Первый прототип Select.**

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, S> selector);

Этот прототип Select принимает входную последовательность и делегат метода-селектора в качестве входных параметров, а возвращает объект, который при перечислении проходит по входной последовательности и выдает последовательность элементов типа S. T и S могут быть как одного, так и разных типов.

При вызове Select делегат метода-селектора передается в аргументе selector. Метод-селектор должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа S. Операция Select вызовет метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Метод-селектор выберет интересующую часть входного элемента, создаст новый элемент — возможно, другого типа (даже анонимного) — и вернет его.

**Второй прототип Select.**

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, int, S> selector);

В этом прототипе операции Select методу-селектору передается дополнительный целочисленный параметр. Это индекс, начинающийся с нуля, входного элемента во входной последовательности.

**Take.**

Операция Take возвращает указанное количество элементов из входной последовательности, начиная с ее начала. Операция Take имеет один прототип, описанный ниже:

public static IEnumerable<T> Take<T>(

this IEnumerable<T> source,

int count);

Этот прототип указывает, что Take принимает входную последовательность и целое число count, задающее количество элементов, которые нужно вернуть, и возвращает объект, который при перечислении выдает первые count элементов из входной последовательности.

Если значение count больше количества элементов во входной последовательности, тогда каждый элемент из нее попадает в выходную последовательность.

**OrderBy.**

Операция OrderBy позволяет упорядочить входную последовательность на основе метода keySelector, который возвращает значение ключа для каждого входного элемента. Упорядоченная выходная последовательность IOrderedEnumerable<T> выдается в порядке возрастания на основе значений возвращенных ключей. Сортировка, выполненная операцией OrderBy, определена как неустойчивая. Это значит, что она не сохраняет входной порядок элементов.

Операция OrderBy имеет два прототипа.

**Первый прототип OrderBy.**

public static IOrderedEnumerable<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector)

where

К : IComparable<K>;

В этом прототипе операции OrderBy передается входная последовательность source и делегат метода keySelector, а возвращается объект, который при перечислении проходит входную коллекцию source, собирая все элементы и передавая каждый из них методу keySelector, таким образом, извлекая каждый ключ и упорядочивая последовательность на основе этих ключей.

Методу keySelector получает входной элемент типа T и возвращает поле внутри элемента, которое используется в качестве значения ключа типа К для этого входного элемента. Типы К и T могут быть одинаковыми или разными. Тип значения, возвращенного методом keySelector, должен реализовывать интерфейс IComparable.

**Второй прототип OrderBy.**

public static IOrderedEnumerable<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector,

IComparer<K> comparer);

Этот прототип такой же, как первый, за исключением того, что он позволяет передавать объект-компаратор. Если используется эта версия операции OrderBy, то нет необходимости в том, чтобы тип K реализовывал интерфейс IComparable.

**Join.**

Операция Join выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей, извлеченных из каждого элемента этих последовательностей.

public static IEnumerable<V> Join<T, U, K, V>(

this IEnumerable<T> outer,

IEnumerable<U> inner,

Func<T, K> outerKeySelector,

Func<U, K> innerKeySelector,

Func<T, U, V> resultSelector);

Первый аргумент метода имеет имя outer. Поскольку это расширяющий метод, последовательность, на которой вызывается операция Join, будет называться внешней последовательностью.

Операция Join возвращает объект, который при перечислении сначала проходит последовательность inner элементов типа U, вызывая метод innerKeySelector по одному разу для каждого элемента и сохраняя элемент, на который ссылается его ключ, в хеш-таблице. Затем он проходит последовательность outer элементов типа T. По мере того, как возвращаемый объект перечисляет каждый объект последовательности outer, он вызывает метод outerKeySelector для получения ключа и извлекает соответствующий элемент последовательности inner из хеш-таблицы, используя этот ключ.

**Операция GroupBy.**

Операции группирования помогают объединять вместе элементы последовательности по общему ключу. Операция GroupBy используется для группирования элементов входной последовательности.

Все прототипы операции GroupBy возвращают последовательность элементов IGrouping<K, Т>.

Здесь IGrouping<K, Т> - интерфейс, который определен, как показано ниже:

public interface IGrouping<K, Т> : IEnumerable<T>

{

K Key { get; }

}

Таким образом, IGrouping — это последовательность элементов типа T с ключами типа К.

# ===47. Рефлексия. System Type.

**Рефлексия** - это процесс обнаружения типов во время выполнения приложения. Например, рефлексия позволяет извлечь список всех типов, которые содержатся внутри определенной сборки \*.dll или \*.ехе (или даже внутри файла \*.netmodule если речь идет о многофайловой сборке), в том числе методов, полей, свойств и событий, определенных в каждом из них

Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой
* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

Эти классы представляют составные блоки типа и приложения: методы, свойства и т.д. Но чтобы получить информацию о членах типа, нам надо воспользоваться классом **System.Type**.

Класс **System.Type** представляет изучаемый тип, инкапсулируя всю информацию о нем. С помощью его свойств и методов можно получить эту информацию. Некоторые из его свойств и методов:

* Метод FindMembers() возвращает массив объектов MemberInfo данного типа
* Метод GetConstructors() возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
* Метод GetEvents() возвращает все события данного типа в виде массива объектов EventInfo
* Метод GetFields() возвращает все поля данного типа в виде массива объектов FieldInfo
* Метод GetInterfaces() получает все реализуемые данным типом интерфейсы в виде массива объектов Type
* Метод GetMembers() возвращает все члены типа в виде массива объектов MemberInfo
* Метод GetMethods() получает все методы типа в виде массива объектов MethodInfo
* Метод GetProperties() получает все свойства в виде массива объектов PropertyInfo
* Свойство IsAbstract возвращает true, если тип является абстрактным
* Свойство IsArray возвращает true, если тип является массивом
* Свойство IsClass возвращает true, если тип представляет класс
* Свойство IsEnum возвращает true, если тип является перечислением
* Свойство IsInterface возвращает true, если тип представляет интерфейс

Получение типа

Чтобы управлять типом и получать всю информацию о нем, нам надо сперва получить данный тип. Это можно сделать тремя способами:

* с помощью ключевого слова typeof,
  + Type myType = typeof(User);
* с помощью метода GetType() класса Object
  + Type myType = user.GetType();
* применяя статический метод Type.GetType().
  + Type myType = Type.GetType("TestConsole.User", false, true);

# ===48. Классы для работы с файловой системой.

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: **Directory и DirectoryInfo**.

**Класс Directory (СТАТИЧЕСКИЙ)**

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* CreateDirectory(path) - создает каталог по указанному пути path
* Delete(path) - удаляет каталог по указанному пути path
* Exists(path) - определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* GetDirectories(path): получает список каталогов в каталоге path
* GetFiles(path): получает список файлов в каталоге path
* Move(sourceDirName, destDirName): перемещает каталог
* GetParent(path): получение родительского каталога

**Класс DirectoryInfo**

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* Create(): создает каталог
* CreateSubdirectory(path): создает подкаталог по указанному пути path
* Delete(): удаляет каталог
* Свойство Exists: определяет, существует ли каталог
* GetDirectories(): получает список каталогов
* GetFiles(): получает список файлов
* MoveTo(destDirName): перемещает каталог
* Свойство Parent: получение родительского каталога
* Свойство Root: получение корневого каталога

Создание каталога:

string path = @"C:\SomeDir";

string subpath = @"program\avalon";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{ dirInfo.Create(); }

dirInfo.CreateSubdirectory(subpath);

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов **File и FileInfo**. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Методы и свойства класса **FileInfo**:

* CopyTo(path): копирует файл в новое место по указанному пути path
* Create(): создает файл
* Delete(): удаляет файл --- объект\_файла.Delete()
* MoveTo(destFileName): перемещает файл в новое место
* Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
* Свойство DirectoryName: получает полный путь к родительскому каталогу
* Свойство Exists: указывает, существует ли файл
* Свойство Length: получает размер файла
* Свойство Extension: получает расширение файла
* Свойство Name: получает имя файла
* Свойство FullName: получает полное имя файла

Класс **File**: (СТАТИЧЕСКИЙ) - т.е. не привязан к конкретному файлу:

* Copy(): копирует файл в новое место - File.Copy(path)
* Create(): создает файл
* Delete(): удаляет файл --- File.Delete(path)
* Move: перемещает файл в новое место
* Exists(file): определяет, существует ли файл

Получение информации о файле

string path = @"C:\apache\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

Console.WriteLine("Имя файла: {0}", fileInf.Name);

Console.WriteLine("Время создания: {0}", fileInf.CreationTime);

Console.WriteLine("Размер: {0}", fileInf.Length);

}

# ===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.

Интерфейс *IDisposable* объявляет один единственный метод *Dispose*, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

Синтаксис C# также предлагает синонимичную конструкцию для автоматического вызова метод *Dispose* - конструкцию **using**:

*using (Font font1 = new Font("Arial", 10.0f))*

*{*

*byte charset = font1.GdiCharSet;*

*}*

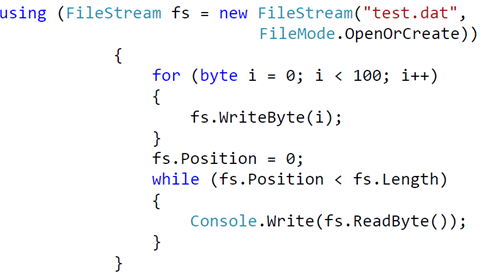
Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

* Свойство Length: возвращает длину потока в байтах
* Свойство Position: возвращает текущую позицию в потоке
* Метод Read: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
* Метод long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* Метод Write: записывает в файл данные из массива байтов. Принимает три параметра: Write(byte[] array, int offset, int count)
  + array - массив байтов, откуда данные будут записываться в файла
  + offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  + count - максимальное число байтов, предназначенных для записи

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. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.



*Поток данных* – это абстрактное представление данных в виде последовательности байт.

Классы для работы с потоками, связанными с хранилищами:

* **FileStream** – класс для работы с файлами, как с потоками (System.IO).
* **MemoryStream** – класс для представления потока в памяти (System.IO).
* **NetworkStream** – работа с сокетами, как с потоками (System.Net.Sockets).
* **PipeStream** – абстрактный класс из пространства имён System.IO.Pipes, базовый для классов-потоков, которые позволяют передавать данные между процессами операционной системы.

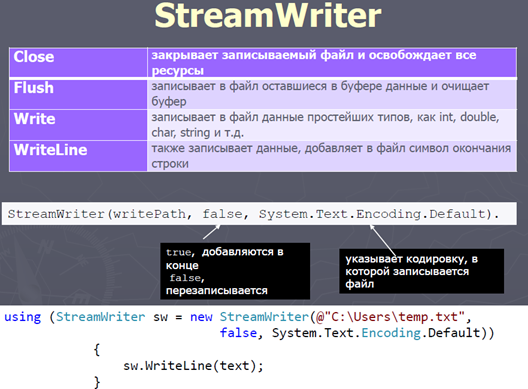
# ===50. Классы адаптеры потоков.

Адаптеры потоков служат оболочкой потока, преобразуя информация определенного формата в набор байтов (сами адаптеры потоками не являются).

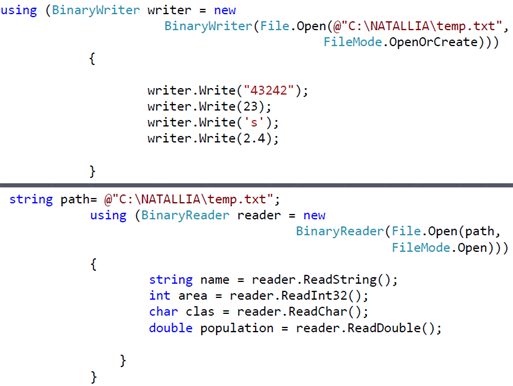
* Текстовые адаптеры (для типов string и char):
  + TextReader
  + TextWriter
  + StreamReader
    - Close - закрывает считываемый файл и освобождает все ресурсы
    - Peek - возвращает следующий доступный символ или -1
    - Read - считывает и возвращает следующий символ в численном представлении.
    - ReadLine - считывает одну строку в файле
    - ReadToEnd - считывает весь текст из файла



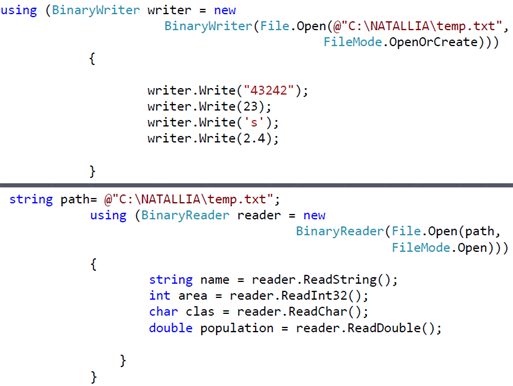
* + StreamWriter
    - Close - закрывает записываемый файл и освобождает все ресурсы
    - Flush - записывает в файл оставшиеся в буфере данные и очищает буфер
    - Write - записывает в файл данные простейших типов, как int, double, char, string и т.д.
    - WriteLine - также записывает данные, добавляет в файл символ окончания строки



* + StringReader
  + StringWriter
* Двоичные адаптеры (для типов int, bool, string и float):
  + BinaryReader
    - Close - закрывает поток и освобождает ресурсы
    - ReadBoolean - считывает значение bool и перемещает указатель на один байт считывает один байт и перемещает указатель на один байт
    - ReadChar - считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
    - ReadDecimal - считывает значение decimal и перемещает указатель на 16 байт
    - ReadDouble - считывает значение double и перемещает указатель на 8 байт
    - ReadInt16 - считывает значение short и перемещает указатель на 2 байта
    - ReadInt32 - считывает значение int и перемещает указатель на 4 байта
    - ReadInt64 - считывает значение long и перемещает указатель на 8 байт
    - ReadSingle - считывает значение float и перемещает указатель на 4 байта
    - ReadString - считывает значение string. Каждая строка предваряется значением длины строки



* + BinaryWriter
    - Close - закрывает поток и освобождает ресурсы
    - Flush - очищает буфер, дописывая из него оставшиеся данные в файл
    - Seek - устанавливает позицию в потоке
    - Write - записывает данные в поток



* Адаптеры XML:
  + XmlReader
  + XmlWriter