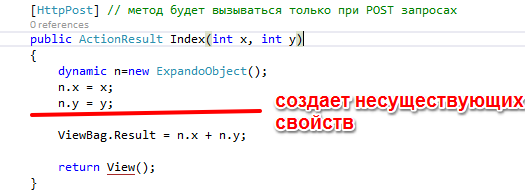
Если свойства нет то добавляет его, работате как в javascript



ViewBag is of type dynamic but, is internally an System.Dynamic.ExpandoObject()

It is declared like this:

dynamic ViewBag = new System.Dynamic.ExpandoObject();

which is why you can do :

ViewBag.Foo = "Bar";

A Sample Expander Object Code:

public class ExpanderObject : DynamicObject, IDynamicMetaObjectProvider

{

public Dictionary<string, object> objectDictionary;

public ExpanderObject()

{

objectDictionary = new Dictionary<string, object>();

}

public override bool TryGetMember(GetMemberBinder binder, out object result)

{

object val;

if (objectDictionary.TryGetValue(binder.Name, out val))

{

result = val;

return true;

}

result = null;

return false;

}

public override bool TrySetMember(SetMemberBinder binder, object value)

{

try

{

objectDictionary[binder.Name] = value;

return true;

}

catch (Exception ex)

{

return false;

}

}

}

Данный способ позволяет вызывать нестатические челены через статическое одно поле (самоассациация) GameChat.Instance.SendChatMessage(playerName + " завалил монстра!");

public class GameChat : Photon.MonoBehaviour

{

private static GameChat \_instance;

public static GameChat Instance{get { return \_instance; }}

private readonly List<string> \_messages = new List<string>();

private string \_messageText = "";

private bool \_showInput;

void Awake()

{

\_instance = this;

}

public class My

{

private static My \_my;

public static My M

{

get

{

return \_my ?? (\_my = new My());

}

}

public string Name { get; set; }

}

.dll

загружается в память executable и в дальнейшем по мере необходимости обращается к нему

А теперь посмотрим на версию этого метода, которая написана с интернирова-

нием строк:

private static Int32 NumTimesWordAppearsIntern(String word, String[]

wordlist) {

// В этом методе предполагается, что все элементы в wordlist

// ссылаются на интернированные строки

word = String.Intern(word);

Int32 count = 0;

for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {

if (Object.ReferenceEquals(word, wordlist[wordnum]))

count++;

}

return count;

}

Во-первых, в этой версии экономится память, если

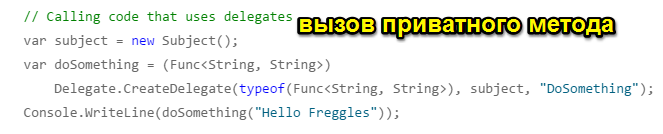
слово повторяется в списке слов, потому что теперь wordlist содержит много-

численные ссылки на единственный объект String в куче. Во-вторых, эта версия

работает быстрее, потому что для выяснения, есть ли указанное слово в массиве,

достаточно простого сравнения указателей.

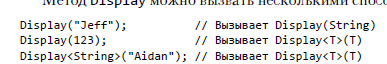
Вызов приватного метода без явной рефлексии



Перебор элементов

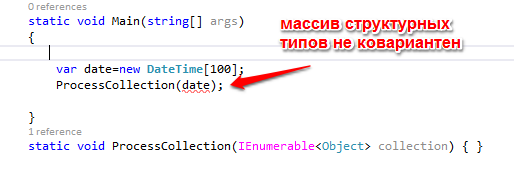
Array.ForEach(parentTask.Result, Console.WriteLine));

Выбор вызываемого метода



НЕЛЬЗЯ

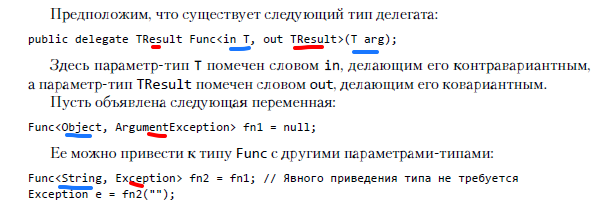




Общая информация

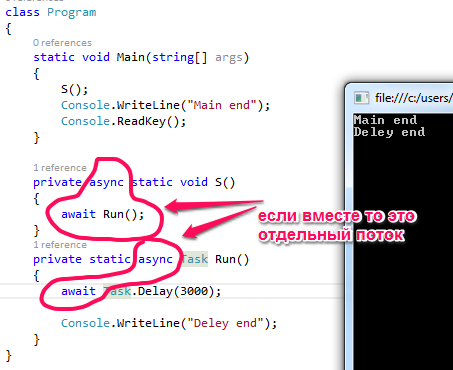
List T — внутренне хранит массив. Добавление нового элемента это либо установление значения в массиве, либо копирование существующего массива в новый, который в два раза больше (недокументированно) и только потом установке значения. Удаление элемента из List T требует копирования расположенных за ним элементов на позицию назад. По индексу RemoveAt() удалять значительно быстрее чем по значению Remove() (происходит сравнение каждого элемента где бы он не находился).  
  
Массивы всегда фиксированы по размеру, но изменяемы в терминах элементов.  
  
LinkedList T — связанный список, позволяет быстро удалять, вставлять новые элементы, нет индекса, но проход по нему остается эффективным.  
  
ReadOnlyDictionary — просто оболочка, которая скрывает все изменяемые операции за явной реализацией интерфейса. Можно изменять элементы через переданную в ее основу коллекцию.

Контрвариантность и ковариантность



Упрощение обобщений

using DateTimeList = System.Collections.Generic.List<System.DateTime>;



Проверка в статическом конструкторе класса на правильность создания обобщения

internal sealed class GenericTypeThatRequiresAnEnum<T> {

static GenericTypeThatRequiresAnEnum() {

if (!typeof(T).IsEnum) {

throw new ArgumentException("T must be an enumerated type");

}

}

}

Чтобы получить правильную результирующую строку (иначе выведет число (сумма каньюнкции)), тип Actions можно определить и без атрибута [Flags]. Для этого достаточно указать формат "F":

// [Flags] // Теперь это просто комментарий

internal enum Actions {

None = 0

Read = 0x0001,

Write = 0x0002,

ReadWrite = Actions.Read | Actions.Write,

Delete = 0x0004,

Query = 0x0008,

Sync = 0x0010

}

Actions actions = Actions.Read | Actions.Delete; // 0x0005

Console.WriteLine(actions.ToString("F")); // "Read, Delete"

FCL - [.NET Framework Class Library ()](https://msdn.microsoft.com/en-us/library/gg145045%28v=vs.110%29.aspx)

интерфейсы допускают EIMI (explicit interface method implementation – явную реализацию метода интерфейса). IComparable.CompareTo

Интерфейс подразумевает отношение «поддерживает функциональность». Например, тип может преобразовывать экземпляры самого себя в другой тип (IConvertible)

------------

**Простота использования.** Базовый тип может предоставлять массу функций, и в производном типе потребуется внести лишь незначительные изменения, чтобы изменить его поведение. При создании интерфейса в новом типе придется реализовывать все члены.

------------

**Четкая реализация.** Как бы хорошо ни был документирован контракт, вряд ли будет реализован абсолютно корректно

Интерфейс реализующий сравнение но при передаче вызываем другой метод, чтобы не возникала упаковка и распаковка

internal struct SomeValueType : IComparable {

private Int32 m\_x;

public SomeValueType(Int32 x) { m\_x = x; }

public Int32 CompareTo(SomeValueType other) {

return(m\_x \_ other.m\_x);

}

// ПРИМЕЧАНИЕ: в следующей строке не используется public/private

Int32 IComparable.CompareTo(Object other) {

return CompareTo((SomeValueType) other);

}

}

Присвоение делегату метода и возвращаемое значение на случай если null

Func<String> f = () => SomeMethod() ?? "Untitled";

Прочитать и понять эту строку намного проще, чем следующий фрагмент кода,

требующий присваивания переменных и использования нескольких операторов:

Func<String> f = () => { var temp = SomeMethod();

return temp != null ? temp : "Untitled";};

Операторы Nullable Type

Еще C# позволяет применять операторы к экземплярам null-совместимых типов.

Вот несколько примеров:

private static void Operators() {

Int32? a = 5;

Int32? b = null;

// Унарные операторы (+ ++ - -- ! ~)

a++; // a = 6

b = -b; // b = null

// Бинарные операторы (+ - \* / % & | ^ << >>)

a = a + 3; // a = 9

b = b \* 3; // b = null;

// Операторы равенства (== !=)

if (a == null) { /\* нет \*/ } else { /\* да \*/ }

if (b == null) { /\* да \*/ } else { /\* нет \*/ }

if (a != b) { /\* да \*/ } else { /\* нет \*/ }

// Операторы сравнения (<> <= >=)

if (a < b) { /\* нет \*/ } else { /\* да \*/ }

}

Первые буквы в верхнем регистре

private string UpperFirst(string text)  
{  
 return CultureInfo.CurrentCulture.TextInfo.ToTitleCase(text.ToLower());  
}

private string UpperFirst(string format)  
{  
 return format.First().ToString().ToUpper() +   
 String.Join("", format.Skip(1)).ToLower();  
}

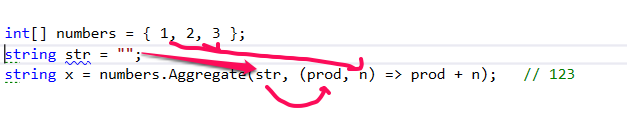
Аuрегация Linq

Вы можете опустить начальное значение (seed value) при вызове метода Aggregate, в таком случае первый элемент входной последовательности *неявно*будет являться начальным значением, и агрегация будет продолжена со второго элемента. Вот предыдущий пример, *без начального значения*:

int[] numbers = { 1, 2, 3 };  
int sum = numbers.Aggregate ((total, n) => total + n);   // 6

Это дает аналогичный результат, но на самом деле, *вычисления*выполняются *другие*. В первом случае мы вычисляли 0 + 1 + 2 + 3; сейчас мы вычисляем: 1 + 2 + 3. Эти различия более показательны при использовании умножения вместо сложения:

int[] numbers = { 1, 2, 3 };  
int x = numbers.Aggregate (0, (prod, n) => prod \* n);   // 0\*1\*2\*3 = 0  
int y = numbers.Aggregate (   (prod, n) => prod \* n);   //   1\*2\*3 = 6



Агрегация и композиция

агрегация

| **Код C#** | |
| --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | **class** B      {  **private** A \_a;  **public** B(A a)*// Объект А живет где-то отдельно (суть не в конструкторе)*          {              \_a = a;          }      } | |  |

композиция

| **Код C#** | |
| --- | --- |
| |  |  | | --- | --- | | 1  2  3  4 | **class** B      {  **private** A \_a = new A();*// Объект А существует только вместе с B*      } | |  |

Конвертер

int[] someArrayYouHaveAsInt=new []{1,1,1};

double[] copyOfArrayAsDouble = Array.ConvertAll<int, double>(

someArrayYouHaveAsInt,

new Converter<int, double>(

delegate(int i) { return (double)i; }

));

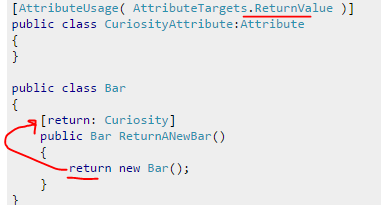
int[] someArrayYouHaveAsInt;

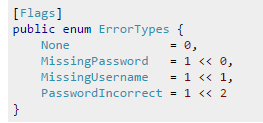
double[] copyOfArrayAsDouble = Array.ConvertAll<int, double>(

someArrayYouHaveAsInt,

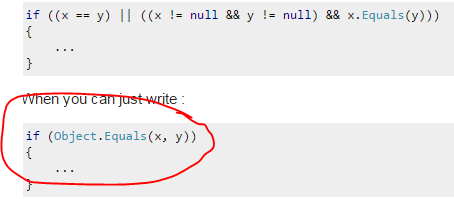
new Converter<int,double>(Convert.ToDouble));

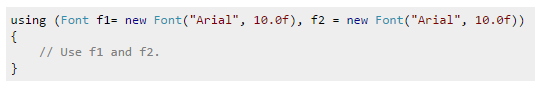
Атрибут для возвращаемого значения



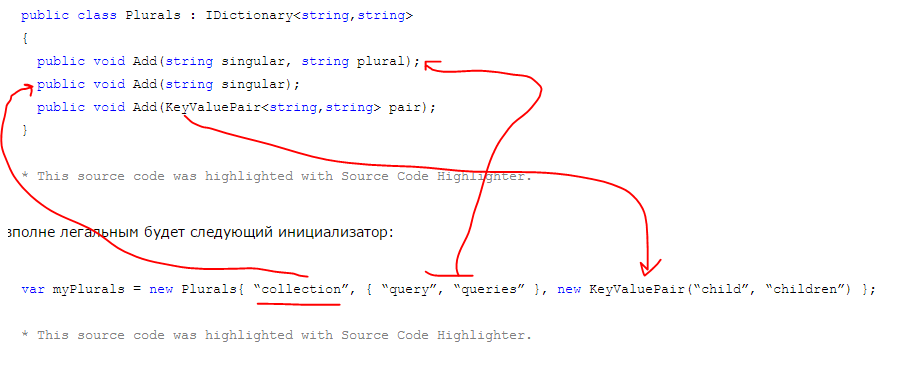


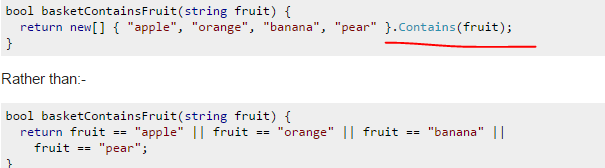












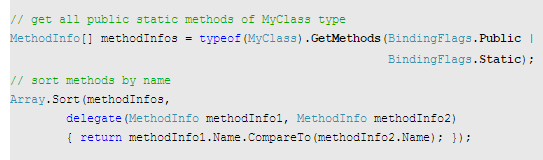
В теле любого метода получаем все о данном методе!!!!

MethodBase.GetCurrentMethod().Name

Создаем экземпляр класса FileInfo и в конструктор передаем в качестве аргумента строку



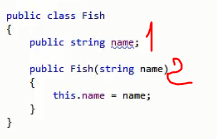
Сортировка массива



Интермирование строки при компиляции (это когда имеются две одинаковые строки)

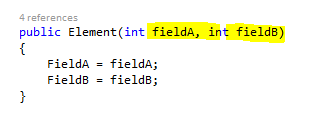
1.Состояние

2.Поведение



ОПЕРАЦИЯ явного приведения работает быстрей чем as

Пользовательский конструктор принимающий два целочисленных аргумента



@ - не является частью идентификатора



Snipets – снипеты

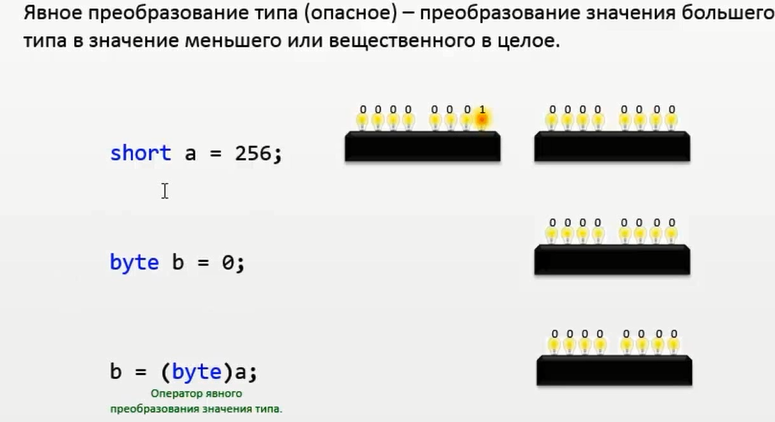
Ctrl+k+x(для быстрого создания конструкций кода)

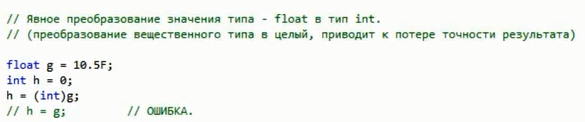
Cast (каст) – приведение типа

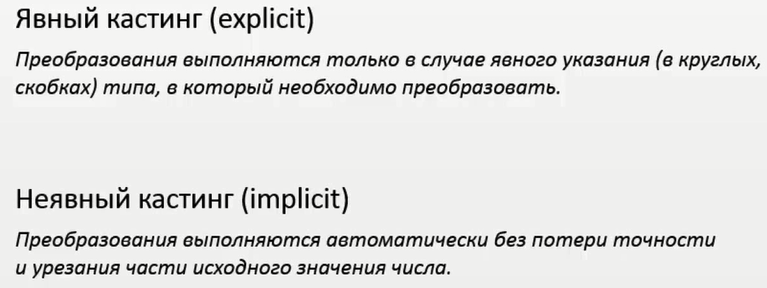
НЕ ПУТАТЬ

Кастинг – преобразование типа

Бывает кастинг безопасный и не безопасный



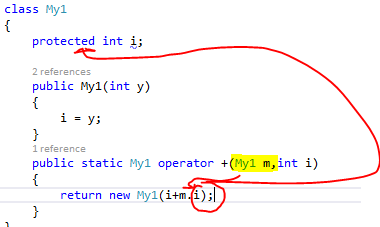




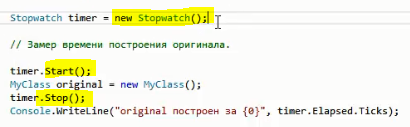
Константа представлена как непосредственное значение или литерал (в рефлекторе)



На закрытые члены класса инкапсуляция не распространяется если они вызываются в теле класса в котором объвлены



StopWatch – класс использующийся для замера времени работы программы

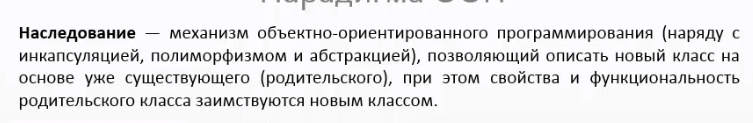


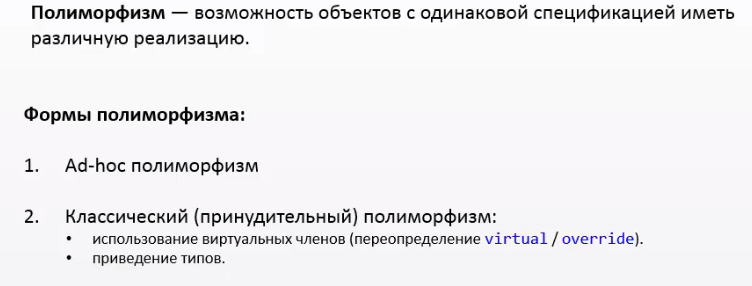


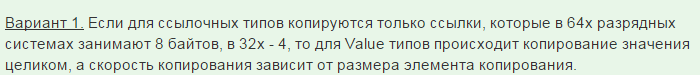
Самоасацияция – когда создаем внутри класса экземпляр этого класса

Инстанцируем class MyClass (иными словами создаем экземпляр класса)







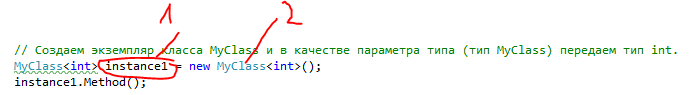


Метод запрещен к дальнейшему переопределению (применяется только к виртуальным методам), может применяться к свойствам





1. Переменная
2. Создается экземпляр класса



Стереотипы языка c#

Классы

Абстрактные классы

Интерфейсы

перечисления

Стек – область памяти для хранения адресов возврата вызываемых процедур

В стеке хранятся все локальные переменные методов

В ссылочных типах - Адрес объекта хранится в стеке но сам объект в куче

Ссылка адрес памяти объекта , экземпляра

Ключевое слово new – создать экземпляр на куче

Паридигмы ООП

Абстракция

Посылка сообщения

Повторное использование

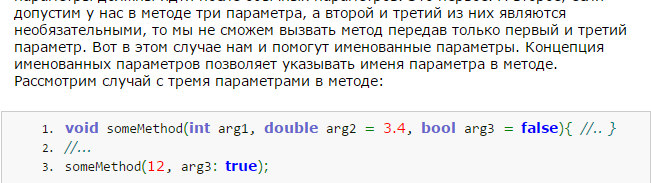
Инкапсуляция

Наследование

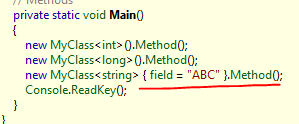
Полиморфизм

Ctrl+k d Выравнивание кода

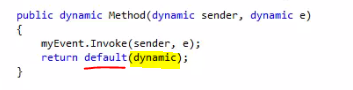
Именованные параметры (опциональные параметры)



Рефлектор оптимизировал код



Вернуть значение для данного типа по умолчанию

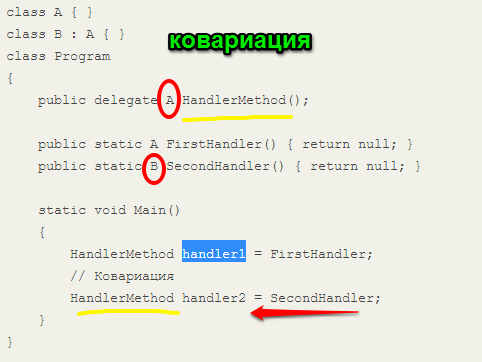


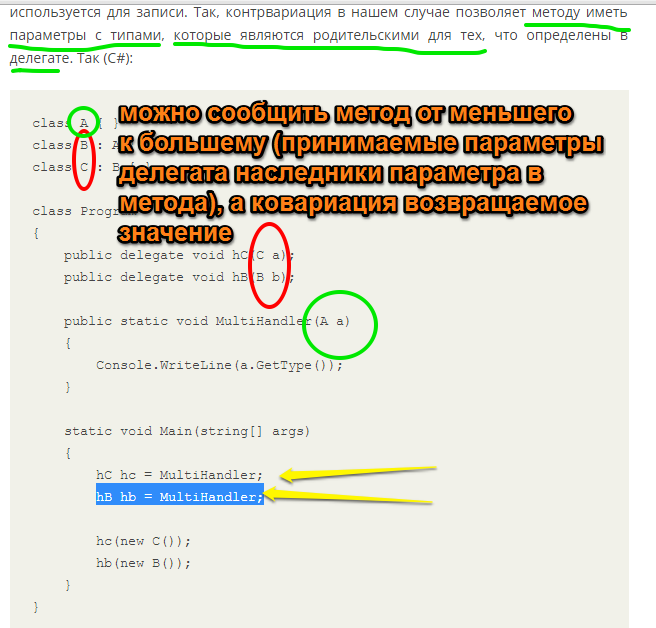
ТОЛЬКО ДВА стереотипа - интерфейс и делегат поддерживают технику ковариантности и контрвариантности

Упкаст (приведение к типу вверх (к родителльскому))

ДаунКаст (приведение вниз)

Делегаты становятся еще более гибкими средствами программирования благодаря двум свойствам: ковариантности иконтравариантности. Как правило, метод, передаваемый делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат. Но в отношении производных типов это правило оказывается не таким строгим благодаря ковариантности и контравариантности. В частности, **ковариантность** позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. А **контравариантность** позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

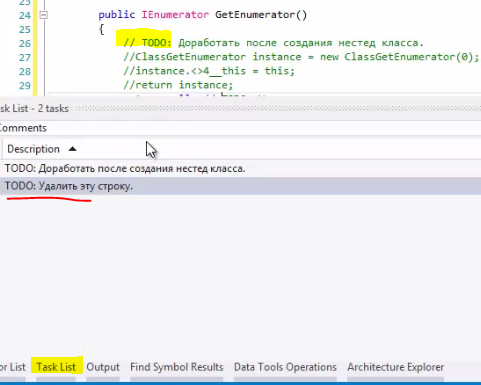




Инициализируем поля в блоке инициализатора во время создания экземпляра объекта

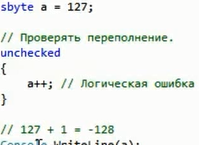
MyClass cl=new MyClass{field = 10};

Если поставить на комментарии TODO то перейдя по вкладке TaskList мы перейдем на строку кода

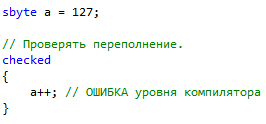


Unchecked – не вызывает ошибку но влияет на результат, он будет ошибочен

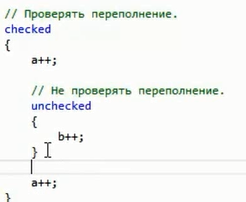
Так же будет и работать и без использования данного ключевого слова



Checked – при переполнении вызывает Exception

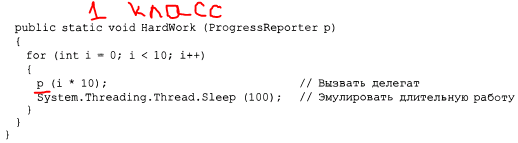


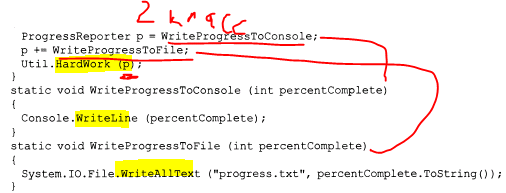
Используются вместе (Unchecked нужен для вложенности)

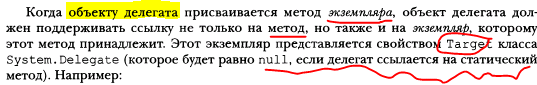


Создали делегат в одном классе, после ему присоединили два метода с другого класса

(одним выводим на экран а вторым пишем в файл)







Перечисления их значения желательно указывать чтобы избежать при битовом умножении и сложении неоднозначности

public enum AllEnum

{ Top=1,

Buttom=2,

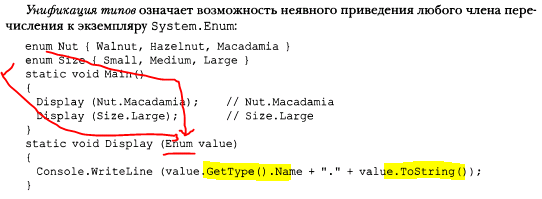
TopBottom=Top|Buttom

}

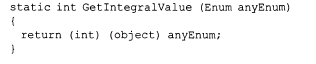
Чтобы корректно можно было осуществить побитовую проверку

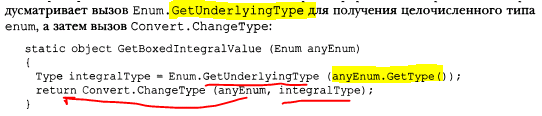
((int)AllEnum.Buttom&(int)AllEnum.TopBottom)

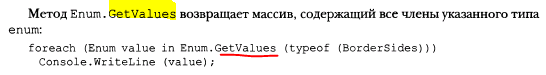
Передача перечислителя в метод



Привести к целочисленному виду

или

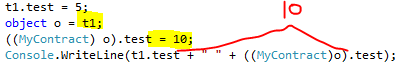
или



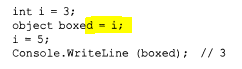
Преобразование перечислений



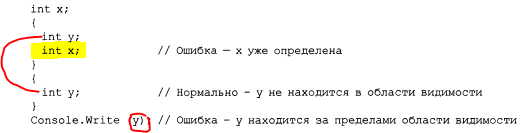
Распаковка упаковка если классы то не копируются



А значения копируются



Область видимости



Присвоение





Выкидывает сообщение если первый аргумент равен ложь

Debug.Assert(3<2,"ошибка");

Запись любых сообщений в файл и на консоль

Trace.Listeners.Add(new TextWriterTraceListener("trace.txt"));

Trace.Listeners.Add(new TextWriterTraceListener(Console.Out));

Trace.AutoFlush = true;

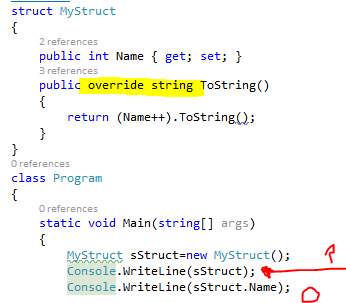
Trace.Write("записываю сообщение в файл и в консоль");

ПРИ ДАННОЙ КОНСТРУКЦИИ НЕ МОЖЕТ ВОЗВРАЩАТЬСЯ NULL, ЭТО ДЕЛАЕТСЯ ПУТЕМ ??

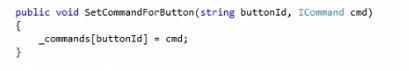


Структура при выводе на консоль без вызова метода ToString() передается копия и значения поля Name не увеличивается, но если вызвать ToString() то значение увеличится!!!!!

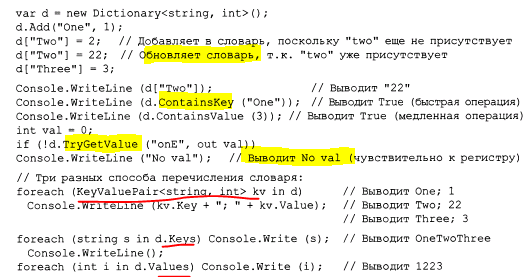
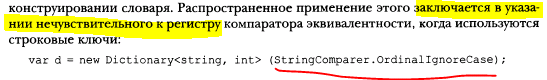
С классами всегда будет увеличиваться, т.к. передается ссылка



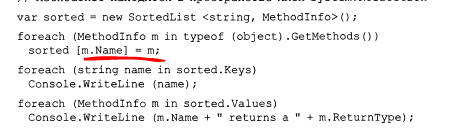
Если ключ не существует то будет добавлен, а если существует то будет перезаписан





,

В созданный список закидываем ключ и значение одновременно



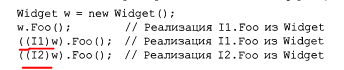
Перекодирование во время чтения из бинарного файла в текстовый формат

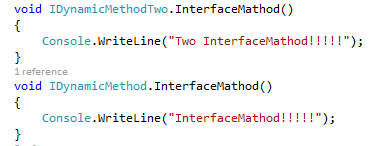






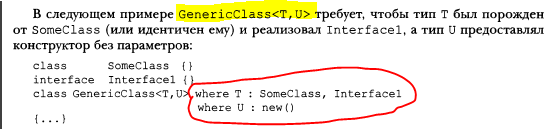
При одинаковых именах в методах интерфейса вызов их методов с приведением типа интерфейса

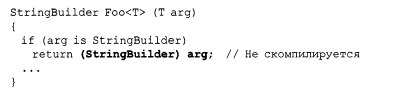




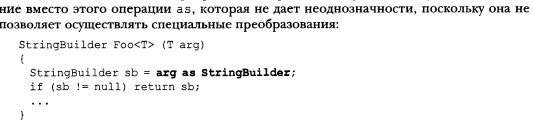


Обобщенные типы ограничения





Нужно через as



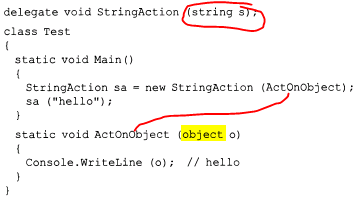
Или



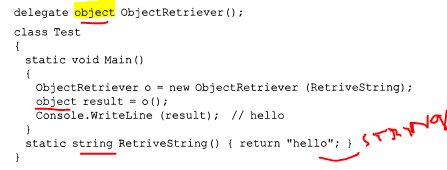
Или



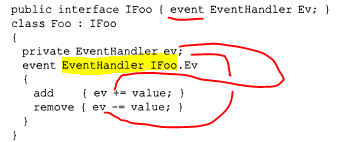
Контрвариантность



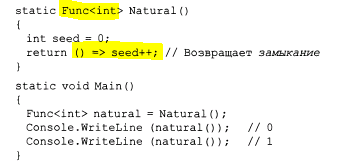
Ковариантность



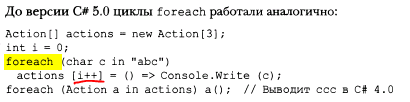
Делегаты объявленный в интерфейсе



Возвращаем лямда выражение и время жизни переменной (путем замыкания) продолжается пока жив делегат



В 5 версии лямда работает иначе

//выведет abc

Блок finally выполняется всегда не смотря на return;

try

{

throw new Exception();

}

catch

{

return;

}

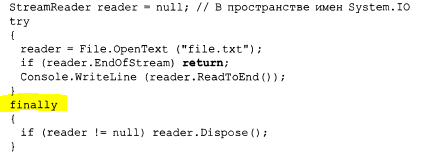
finally

{

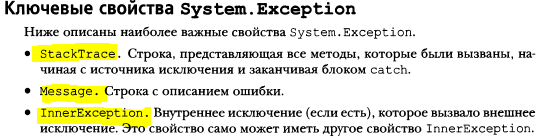
Console.WriteLine("Deleg class {0}", Name);

}

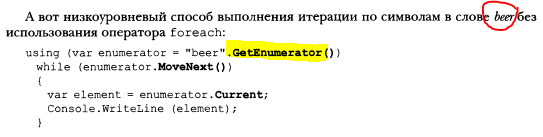
Всегда зайдет в



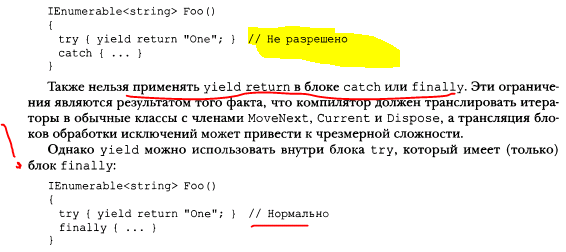
Свойства исключения



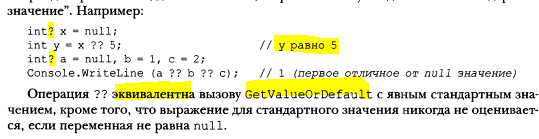
Перечислители



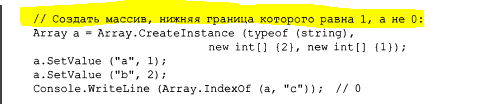
Yield return Разрешения



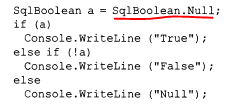
Int? когда null возвращает первое значение не null



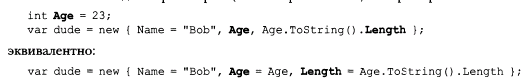
Создание массива начальное значение которого равно 1



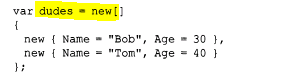
Структура для SQL



Анонимны тип, переменные неявно типизированного типа







Динамическое связывание

Если метода в классе нет, то необходимо переопределить TryInvokeMember чтобы не было ошибки при вызове



public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)

{

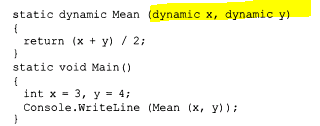
Console.WriteLine(binder.Name+" method call");

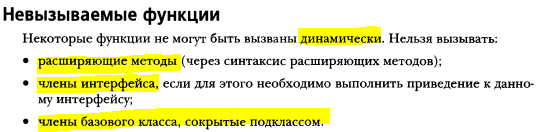
result = null;

return true;

}

Можно сделать и так



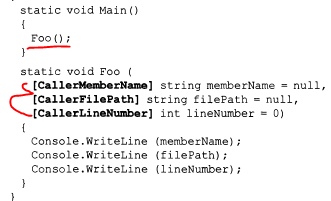




Только так

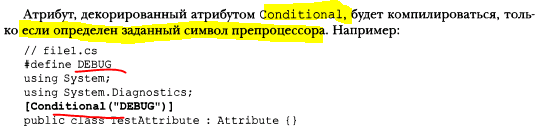


Атрибуты показывающие какой метод вызван, какая строка метода и где находится данный файл



Результат





Если переменная не используется то чтобы не было предупреждения

#pragma warning disable

int y = 6;

#pragma warning restore

Описание метода

/// <summary>

/// описание метода

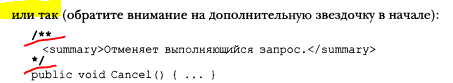
/// </summary>

public void Show()

{

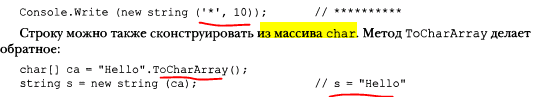
Console.WriteLine("SHOW");

}

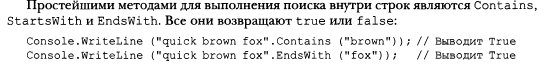


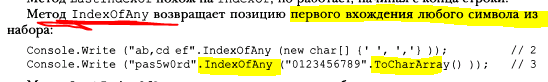
Fxcop программа проверки стилей

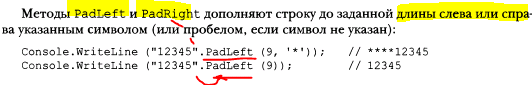
Char

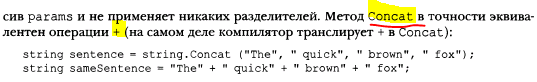
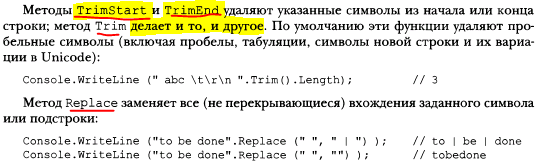


STRING

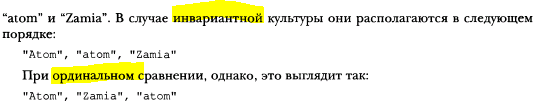








Сравнение строк Equals



Чтобы отсортировать

string str = "Acd";

string str1 = "aad";

List<string> list = new List<string>(){str,str1};

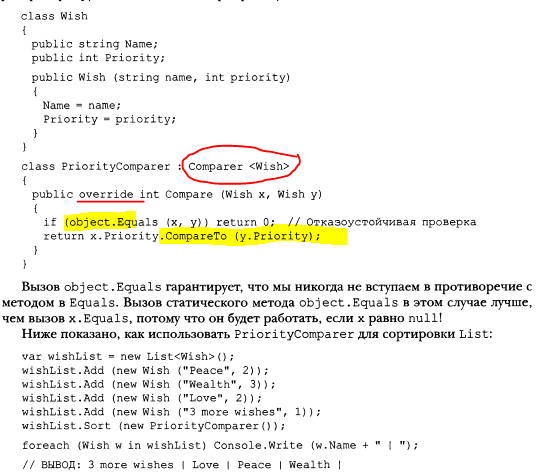
list.Sort(Comparison);

list.ForEach(n => Console.WriteLine(n));

private static int Comparison(string s, string s1)

{

return string.CompareOrdinal(s, s1);}

или сортировка

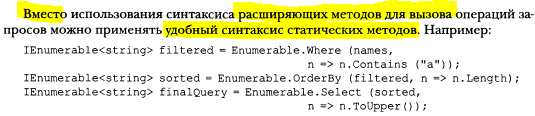
LINQ текущий синтаксис



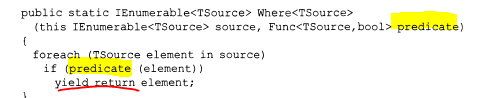
Выражение запроса (синтаксис)



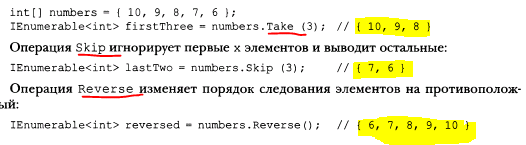
Не очень удобно так как нельзя написать в одной строке

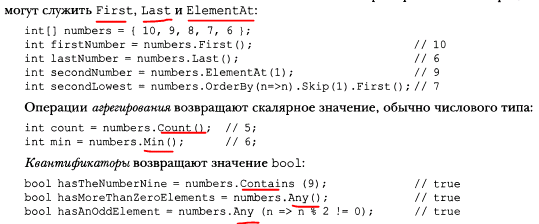


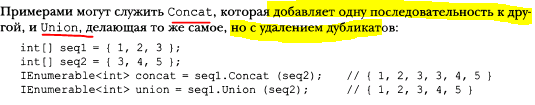
Иными словами Enumerable Where это



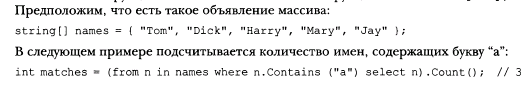
Take, Skip, Reverse и другие запросы







Смешанный синтаксис запроса



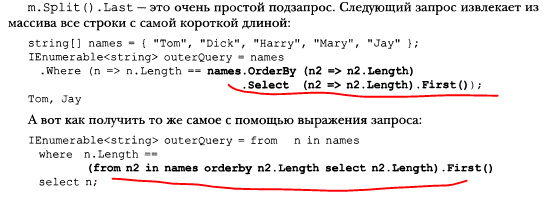
Или можно было так но это длиннее



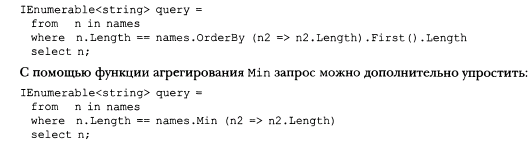
Сортирует по фамилии



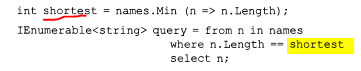
Извлекает все минимальные по длине фамилии !!!



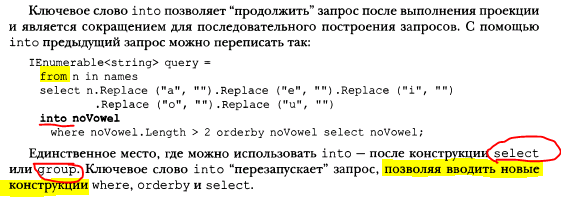
Тоже самое только короче



Но лучше выделить отдельную переменную, т.к. при каждом запросе отрабатывается подзапрос



Into





Выбираем слово и создаем анонимный тип

var query =

from n in names

where n.Contains("ar")

select n

into tmp

where tmp.Length == names.Where(n => n.Contains("ar")).Min(n=>n.Length)

select new {str = tmp,size = tmp.Length};

query.ToList().ForEach(n => Console.WriteLine("query = {0,10}, Size = {1,10}", n.str,n.size));

Выбирает имена в которых присутствуют указанные буквы, удаляет в них эти буквы, а после считает количество символов их, и возвращает целые, не урезанные слова

var names1 = new[] { "Tom", "Dick", "Harry", "Mary", "Jay" }.AsQueryable();

var query0 = from n in names1

let vowelless = n.Replace("a", "").Replace("e", "").Replace("i", "").Replace("o", "").Replace("u", "")

where vowelless.Length > 2

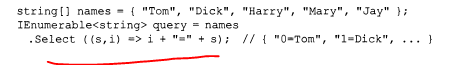
orderby vowelless

select n;

query0.ToList().ForEach(n => Console.WriteLine("query0 = {0,10}", n));

Индексированная фильтрация





TakeWhile Пока не false



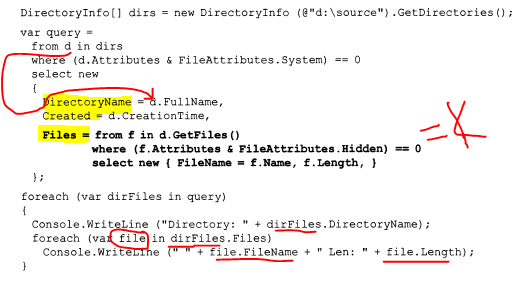
SkipWhile



Distinct удаляет повторения



Коррелированный подзапрос, т.е. он ссылается на объект во внешнем запросе



dynamic возвращает анонимный тип из метода после запроса

static IEnumerable<dynamic> LinqQuery()

{

string[] names = {"Vitek","Vasek"};

var query = from n in names

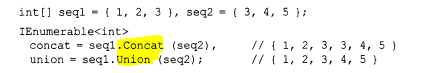
select new {Len=n.Length, Name=n};

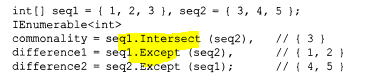
return query;

}

LinqQuery().ToList().ForEach(n=>Console.WriteLine(n.Len+" "+n.Name));

Concat,Exept,Intersect





namespace IEqualityCompar

{

class Program

{

static void Main(string[] args)

{

Contact[] fruits = { new Contact() { Name = "apple" },

new Contact() { Name = "banana" },

new Contact() { Name = "cherry" },

new Contact() { Name = "strawberry" }};

Contact[] fruitsWithLongNames = { new Contact() { Name = "apple-red" },

new Contact() { Name = "banana" },

new Contact() { Name = "cherry-red" },

new Contact() { Name = "strawberry" }};

var stringLengthComparer = new StringLengthEqualityComparer();

IEnumerable<Contact> fruitsWithShortNames = fruits

.Intersect(fruitsWithLongNames, stringLengthComparer);

Console.WriteLine("Using our custom equality comparer:");

foreach (var fruit in fruitsWithShortNames)

{

Console.WriteLine(" - {0}", fruit.Name);

}

Console.ReadKey();

}

}

public class Contact

{

public string Name { get; set; }

public string EmailAddress { get; set; }

}

class StringLengthEqualityComparer : IEqualityComparer<Contact>

{

public bool Equals(Contact x, Contact y)

{

return x.Name == y.Name;

}

public int GetHashCode(Contact obj)

{

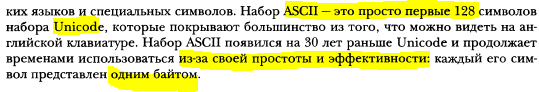
return obj.Name.GetHashCode();

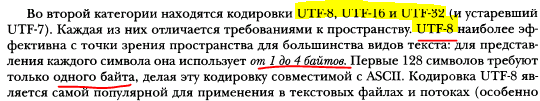
}

}

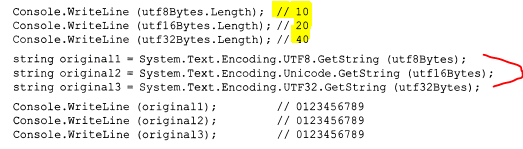
}

UNICODE

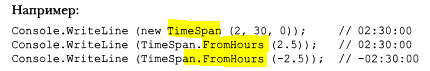


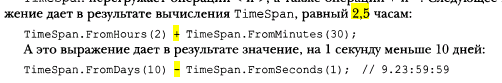


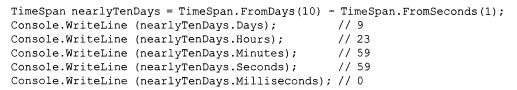
Кодировка в байтовый массив



TimeSpan



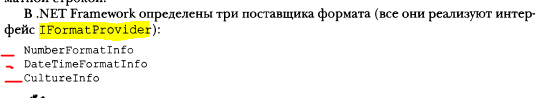




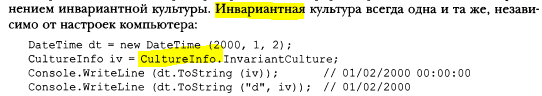
Смещается на3 часа назад



ToString

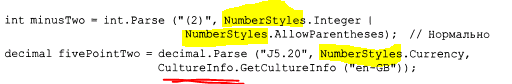








Parse



Округление

Console.WriteLine(35556.333333.ToString("####.#"));

Convert



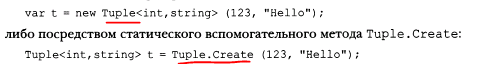


Random более точная реализация, используется в криптографии





Кортежи



IComparable интерфейс



Process запускает консоль со строчкой c ipconfig all

ProcessStartInfo proc = new ProcessStartInfo()

{

FileName = "cmd.exe",Arguments = "/c ipconfig all/",

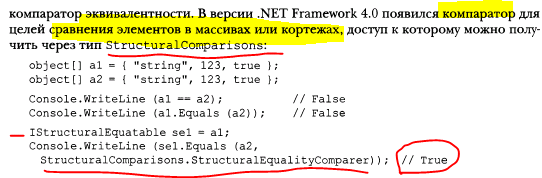
RedirectStandardOutput = true,UseShellExecute = false

};

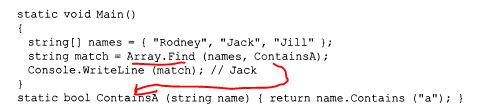
Process pr = Process.Start(proc);

Console.WriteLine(pr.StandardOutput.ReadToEnd());

StructuralsComparisons Сравнение массивов



Array поиск элемента посредством предиката (делегат принимающий объект) и возвращающий правда или ложь



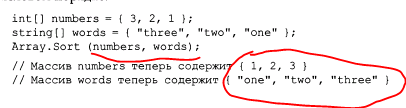
Или

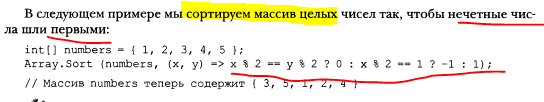


Или

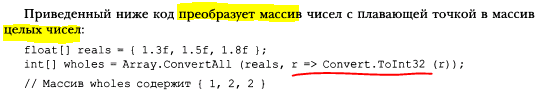


Сортирует второй массив по значениям первого

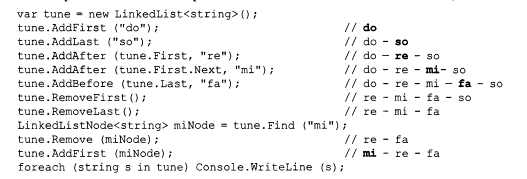




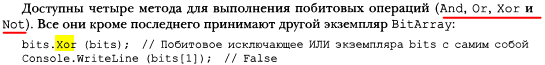
Преобразование массива

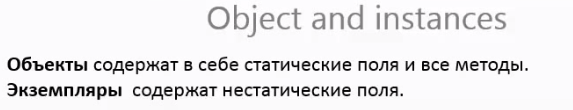


LinkedList двусвязный список

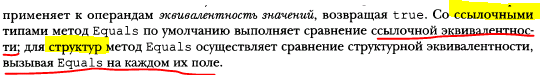


BitArray как правило для массива bool





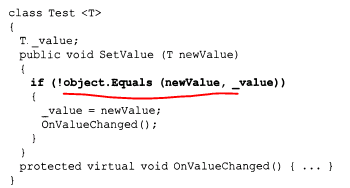
Equals()







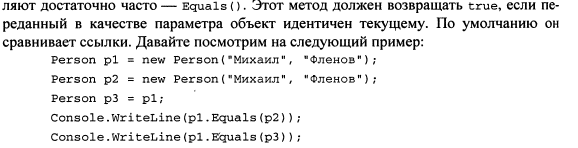
В данном случае статическое сравнение == применить нельзя так как на этапе компиляции не известен обобщенный тип, поэтому нужно применять



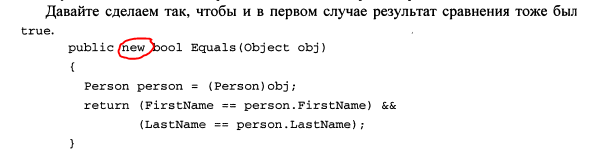
Сравнивает ссылки

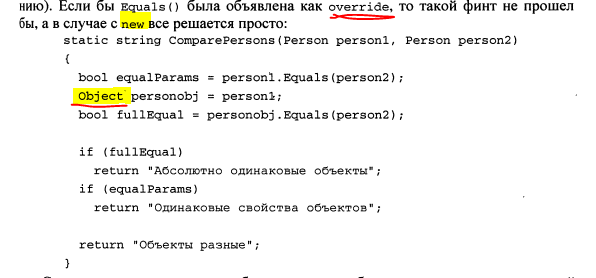






False, false,true





var func = typeof( Convert ).GetMethod( "ToInt32", new Type[] { typeof( **object** ) }, **null** );

**int** val  = (**int**)func.Invoke( **null**, new **object**[] { "123" } );

var i = new {test=10,retg="10"};

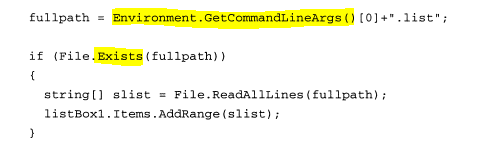
MethodInfo d = typeof(Convert).GetMethod("ToInt32",new Type[]{typeof(object)},null);

MethodInfo p = typeof(Int32).GetMethod("Parse", new Type[] { typeof(string) }, null);

int y=(int) d.Invoke(null,new[] {i.retg});

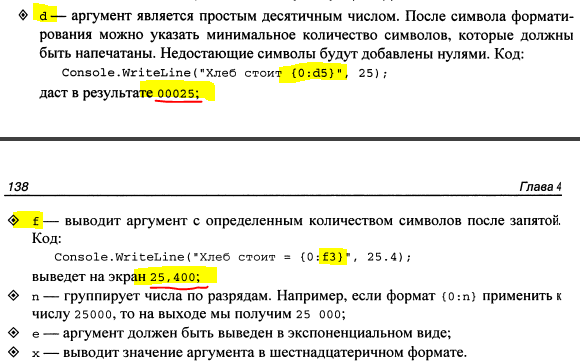
int z=(int) p.Invoke(null,new[] {i.retg});

Console.WriteLine("{0},{1},{2}",y,i.test,z);

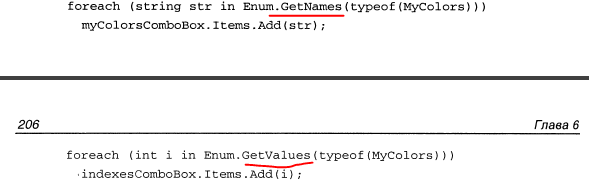


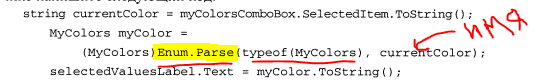


Console()

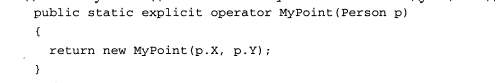


Enum

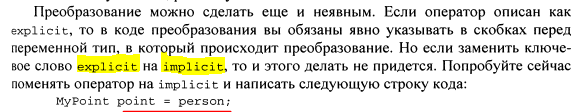




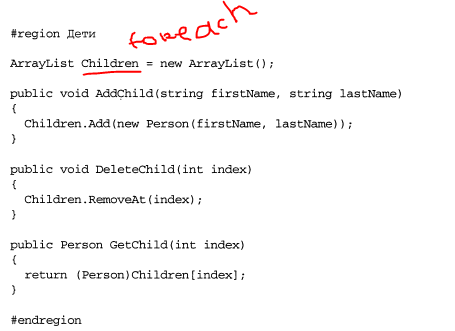
Implicit Explicit

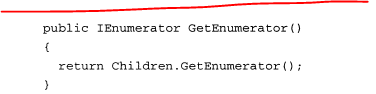


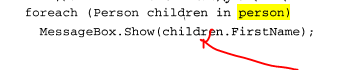




IEnumereble







Exeption

