Полезни неща от курса по ООП с C# .NET

Седмица 1:

Shortcuts:

Rename var/func/class/whatever -> CTRL + R CTRL + R

Copy Line -> CTRL + D

Mark whole line HOME SHIFT + END

Mark whanted area - hold ALT mouse mark

Call Hierrarhy - CTRL + K CTRL + T

Regions - CTR + K, S

Snippets:

cw ctor prop propfull

Как се прави сниппет:

XML file -> .snipped ->

<?xml version="1.0" encoding="utf-8"?>

<CodeSnippets

xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">

<CodeSnippet Format="1.0.0">

<Header>

<Title>

<!--TITLE-->

</Title>

<Shortcut>

<!--SHORCUT-->

</Shortcut>

</Header>

<Snippet>

<Code Language="CSharp">

<![CDATA[ <!--CODE--> $end$]]>

</Code>

<Declarations>

<Literal>

<ID><!--PARAM--></ID>

<Default>true</Default>

<ToolTip><!--INFO--></ToolTip>

</Literal>

</Declarations>

</Snippet>

</CodeSnippet>

</CodeSnippets>

Стил:

Класовете и методи започват с главна буква. Променливи малка (CamelCase)

Всяка променлива на нов ред, по-нагледно, ако трябват коментари.

За WPF приложения:

XAML:

Border

StackPanel

Grid - ColumnDefinitions, ColumnDefinition Width=\*, RowDefinitions, RowDefinition Heigth=\*, Grid.Column, Grid.Row

TextBlock

TextBox

ComboBox - ComboBoxItem

CheckBox

GroupBox

Функционалност на бутоните:

Click=”event\_name” - имаме име на евент, когато натиснем бутона

private void event\_name\_Click(object sender, RoutedEventArgs e) - функция която се изпълнява когато се натисне бутона

(string)((Button) sender).Content //switch

самите елементи могат да имат имена, които във файла, където пишем код на C#, можем да достъпваме, променяме и извличаме данни от/в тях и т.н.

Полезни неща:

TryParse string->int ex. int a = int32.TryParse(str)

Седмица 2:

Полезно:

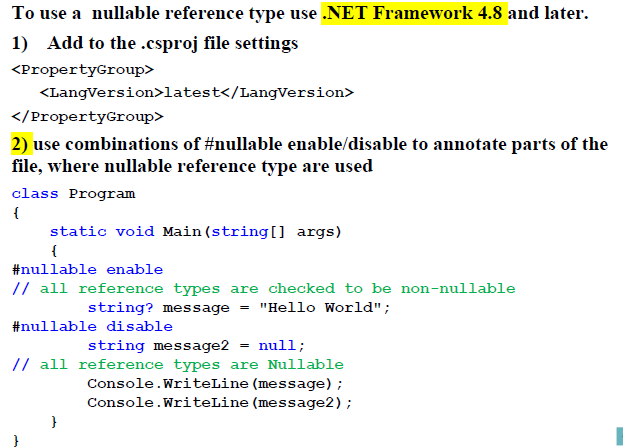
Трябва да пишем пред класовете public.

Нови обекти на клас създаваме <клас-име> обект = new <клас-име>() <- конструктор

Полета и пропъртита - propful tab tab -> field името е с малка започваща, пропартито е същото име с главна започваща. Пропъртито има гет и сет методи, които са със синтаксис

get{...return field\_name} set{...field\_name = value} (value e стойността която се подава при извикване на сет метод). В пропъртито можем да слагаме private пред set / get -> readonly writeonly.

Nullable variables <тип>? име. Това позволява на стойността да бъде null.

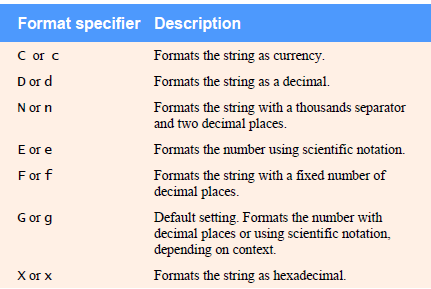


Tuples - когато искаме да пакетираме няколко неща в едно нещо. Например искаме функцията ни да върне наредена твойка инд стринг дабъл -> ползване този тип (int, string double), като пред всяко можем да слагаме имена и след това когато ползваме променливата, която съдържа тези данни да извикваме желаните данни.

“=>” този символ казва, че ако имаме една фунцкия Ф1() директно върни нещо. Като например:

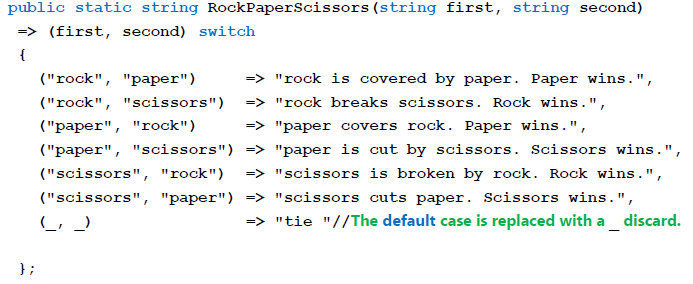
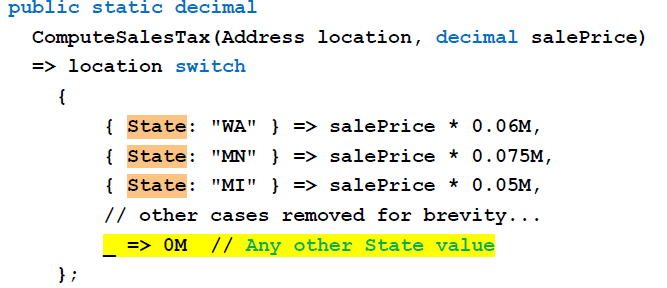
(int dummy, int max, int min) MinMax(int i)

=> (0, int.MinValue, int.MaxValue);



Pattern matching с switch и when. В условията на суйтча можем да кажем <тип> име when (булево условие).

PropertyPattern matching синтаксис:



“is” примерно проверява дали инпута е инт if (input is int count) -> count става стойността на input.

Class Diagram -> Project -> New Item -> Class Diagram

Toolbox добавяме каквото ни е нужно.

Седмица 3

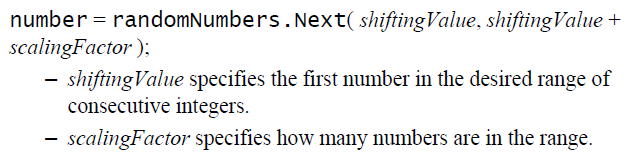
Полезно:

Правим малки смислени функции, които да правят едно нещо, преизползваме код, предодвратяваме повтарящ се код.

Статични методи/променливи. Име на класа . <static\_method/ member>

Random функция.

Random myRandom = new Random() -> myRandom.Next(1, 7) //[1,7)



Множество от константи - enum <name>{EL1, EL2 ...}

private enumMyEnum: *typeName*{*Constant1*,*Constant2*,*...*}

Можем да overload-ваме методи - едно и също име на метод, но различни параметри.

Подаване на аргумент на функция по референция. Можем да използваме думите ref(когато очакваме подадения арг да е инициализиран предварително) и out(когато очакваме, че функцията ще му даде стойност). И когато подаваме аргумент задължително пишем следното func(ref/out varName).

След c#7 можем да извикваме фънк(out int variable) и тази променлива е валидна в скоупа в който е ползвана, ползва се в TryParse

Ако класа, от който искаме да създадем инстанция няма конструктор който да работи с конкретни параметри можем да направим следното

construcotr\_name(<member\_name>: <val>...)

Структура от данни са колекции от дании които са свързани помежду си

Масиви: Lenght, създаваме като <type>[] c = new <type>[number], при създаването на масив всички елементи приемат дефолтни стойности, числовите 0, булевите лъжа, референтните NULL, static Array.Resize(ref myArr, newSize)), ArrayInitializer

foreach (type id in arr){statement}

var когато ползваме тази дума задължително трябва да кажем = някаква стойност иначе ще доведе до грешка, удобно е да се ползва във foreach

Подаване на масив като аргумент на функция става само чрез неговото име без скоби, а в метода трябва да бъде обявено, че параметъра е масив (т.е. със скоби). Подаване без реф прави копие на референцията към която сочи масива, но може чрез нея да се променят елементите

двуметни масиви - int[ , ] myArray = {{1, 2},{3 ,4}} (всеки “ред” има еднакъв брой елементи), но ако си искаме наш си брой правим int[][] myCoolArray{new int[] {2,3 ,3}, new int[] {2,1,2}...}

int[ , ] b; b= new int[3,5], int[][] c; c = new int[2][]; c[0] = new int[5].....

GetLength...или два вложени foreach-а един по редове и другия колони

params (на последно място) пример foo(params double[] numbers)

Седмица 4

Полезно:

Масиви - даваме име на масива в множествено число

\* int size = int.Parse(Console.ReadLine())

масив с размер 0 не е нулл

var names = new[]{........} var e без []

Масивите имат функции и пропъртита от клас Array .NET Framework

-Length

Не можем да променяме данните на масив в foreach трябва for

При копиране на масиви, масивите са референтни типове така, че когато направим int[] a1.. int[] a2 = a1 имаме 2 референции към едно и също нещо. Ако искаме да копираме данните правим int[] anew = new int[aold.Length] и завъртаме един for. Може и с aold.CopyTo(anew,0)

Collections <3:

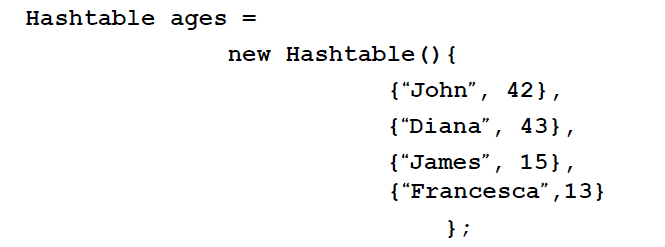
ArrayList - Remove(само пренарежда елементите),Add(ако се наложи се преоразмерява), Insert(ако се наложи се преоразмерява), [] (но не е препоръчителен заради лоша производителност, ползва се за хетерогенни колекции) вместо това->

List - ползва се за хомогенни колекции

Queue(FIFO) - at the back Enqueue, Dequeue (с един цикъл може да я изпразним queue.Count > 0 Dequeu), Dequeue-то връща това, което декювва

Стек(FILO) - push, pop

HashTable - асоциативен списък има ключ/стойност, реализира се с 2 обекта от тип масив, един за ключовете другия за стойноститте които мапваме, не можем да имаме дублиращи се ключове, ContainsKey, (key value)



Седмица 5

Полезни неща:

using System.Linq;

1) избираме from x in SOMETHING

2) условеие where x (изпълнява някакво условие)

2.1) orderby x {decending обратен ред}

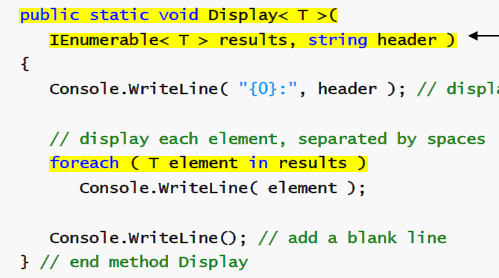
3) избираме select x (избери тези x)

връща се тип IEnumerable<T> - който предлага функционалност з всеки обект който може да се итерира и представя методи за достъп до елементите си

фунцкията Any() към някое IEnumerable дава Т ако има поне един елемент. Ползваме го при проверки дали имаме елементи в това което разглеждаме. Имаме и функция First() която ни дава първия елемент на резултата от заявката ни. Имаме Distinct метод, който маха повтарящи се резултати. В select часта можем да създаваме нови обекти като :



и една дисплей функция:



можем да ползваме let клауза за да създадем нова диапазонна пром. която да съхранява временно резултат ползвам по късно в заявката

Ламбда изрази: можем да ги използваме за създаване на анонимни променливи или делегати. Става като аргументите са от ляво на оператор =>ако има такива и слагаме тялото на “функцията” ни от дясно. Ползва се в LINQ като най често -> (input params) => expression. Повече от един параметъра се слагат в скоби и се разделят със запетайки. Понякога не може да се идентифицират типовете на променливите, затова можем да направим така : (int x, string y) => expression, () => somemethod()

**System.Linq.Queryable**

за Count метод, приема някакво условие и брои елементите вътре в структурата, които удовлетворяват това условие. Имаме TakeWhile().

Select(), Where().Select(), GroupBy - Key, Count

order smth by smth.smth into tempKeyword

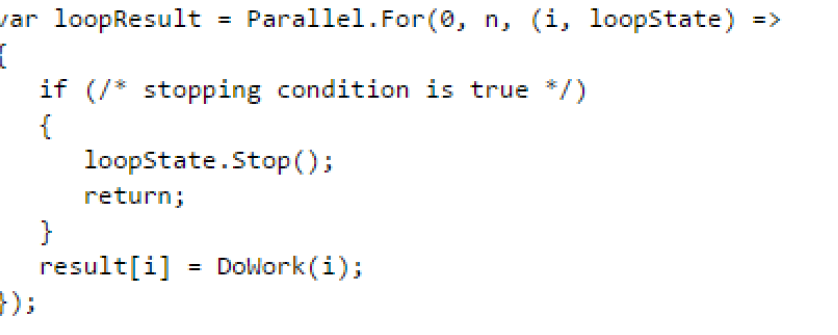
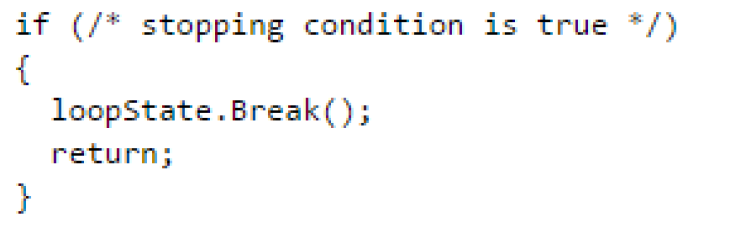
TPL: Task parallel library

**System.Threading.Tasks.Parallel**

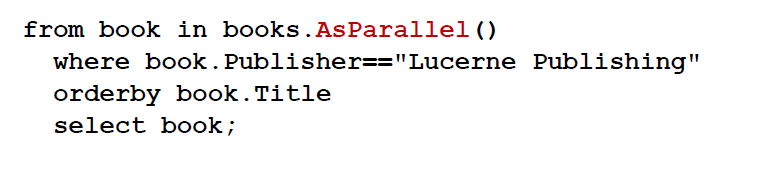
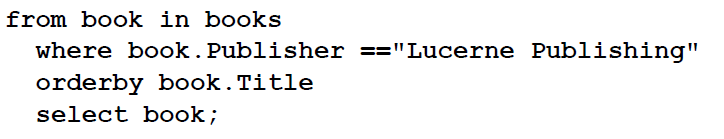
Parralel.For/Foreach

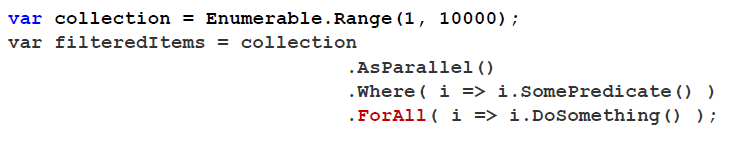
Parralel.Foreach(srcColl, itm =>Process(item)) - операциите в ламбда израза трябва да бъдат независими

Паралелните for, foreach са методи и в тях ползваме Break() - завърши всички операции на всички нишки които са приоритени към текущата итерация на текущата нишка и излез от цикъла, Stop() спри всички итерации когато е възможно.



PLINQ - AsParallel



ForAll

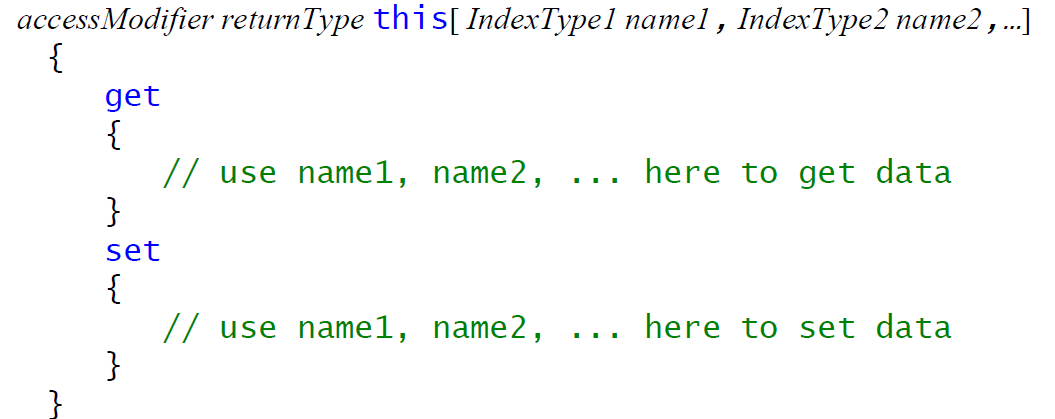
yield - яко нещо

Седмица 6

Полезни неща:

указател this - референция към класа и може да се ползва в нестатични методи

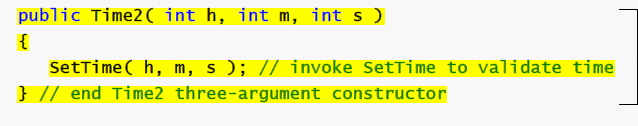
индексъри indexers - класове който инкапсулират списъци с данни може да използват this за да дефинират индексъри, които дани позволят да имаме нещо подобно на масивен-достъп до тези елементи на класас. Индексърите могат да връщат всякакъв тип. Изисвкат this за да се дефинират.

ind tab tab -> 

според това какъв е типа на индекса IndexType се изпълняват различни неща. Важно е да проверяваме дали индексите ни са валидни. Ползват се като направим обект на класа и използваме дума с квадратни скоби и индекс в тях.

оверлоадване на конструктори <3

general purpouse - той е най-сложен (за простия случай)



default constructor objName () : this(... , ... , ...) - инициализатор

copy constructor objName (objName1) : this(objName1.1 .... objName1.2)

\*по този начин ако се наложи промяна някъде на някакъв тип или на нещо в конструктора за всеобщо ползване, само на едно място ще се наложи да направим промяна

S.O.L.I.D ПРИНЦИПИ - Първите 5 принципа на ООП дизайн

1) Single-responsibility principle - един клас трябва да има само една единствена причина да се променя, което значи че всеки клас има една работа!!!! (например клас който изпълнява Х У функции може да се промени първо защото нещо е станало с Х функцията на класа и второ защото нещо се е промени в У функцията на класа, което са 2 неща за което е отговорен класа и нарушава 1ви принцип)

2) Open-closed principle - обектите трябва да бъдат отворени за разширение, но затворени за модификация, лесно да се разширява без да се налага да се модифицира

3) Liskov substitution principle - q(x) да е проперти на обект х от клас Т. Тогава q(y) трябва да работи за обект у от тип S където S e подтип на T. Всеки подклас/наследен клас трябва да може да се субституира от техния базов клас.

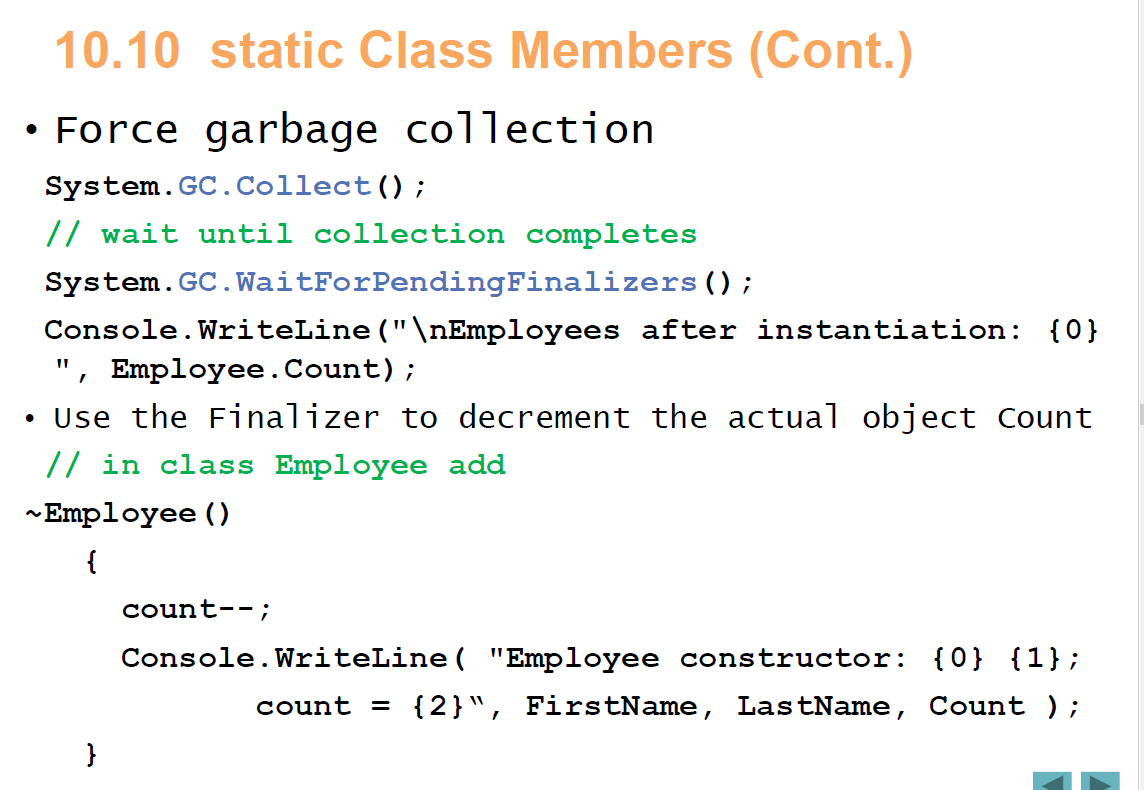
4) Interface segregation principle - не трябва клиентите да имплементират интерфейс или да бъдат принудени даползват методи които не позлват. Класа не трябва да има интерфейс, ако семантиката му не поддържа функционалността му.

5) Dependency Inversion principle - обектите трябва да зависят от абстракция не от конкреция?. Високониво модул не трябва да зависи от ниско ниво модул, но трябва да зависят от абстракция.

HAS-A (Има) - композиция, референтни типове - изменяеми и неизменяеми.

immutable- without set, all fields readonly, private, подкласовете да не променят методи, sealed, никога да не съхраняваме референции към външни изменяеми. ако се налага създаваме копия на теиз неща. Важно и като запазваме в такива неизменяеми да си имаме копи конструктор за техния тип и в пропъртитата да връщаме техни копия а не самите те

за гарбиг колектора



readonly - само с главни букви, като константите, може да не се инициализират при декларирането им, става непроменима след като конструктора завърши изпълнение, т.е. можем да си задаавме стойности в конструктора за тях.

ВСЕКИ КОНСТУРКТОР ТРЯБВА ДА ИНИЦИАЛИЗИРА РЕАДОНЛИ ДАННИТЕ

Как да си направим библиотека:

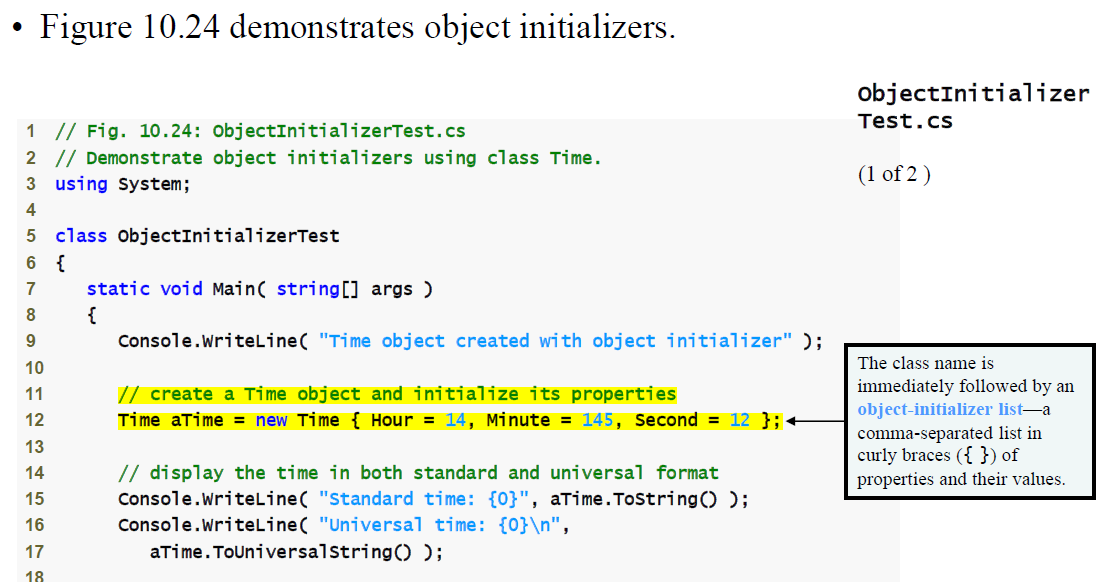
1) всички класове вътре трябва да бъдат публични

2)избираме неймспейс декларация

3)компилираме класа като клас библиотека

4)добавяме референция към нашия клас библиотека solution explorer -> add reference

5)добавяме кои неймспейсове ще ползваме от библиотеката



инициализатор на обект - {пропъртита и стойности за тях, всяко пропърти само по един път, не може да е празен, изпълнява се в реда в който са наредени нещата - първо се вика конструктора на обекта, така, че всяка стойност неупомената тук да присвои ст-ст от конструктора}

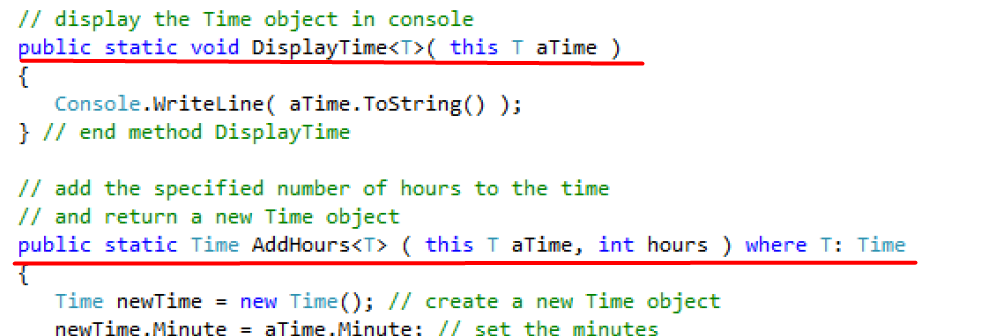
разширяване на клас с функция:

функцията трябва да е като нормална функция, но на първия си аргумент се приема параметъра this, т.е. трябва да има поне един параметър, дефинират

как го правим:

static клас <иметонакласа>Extensions и вътре пишем функциите:

public static void ......<T> (this T smth)



делегати:

delegate - обект, който държи референция към метод, методите като данни, може да присвояваме променливи като функции и да ги подаваме през други методи. Можем и да виками методи чрез променливи от делегат тип.

public delegate <type\_na\_func> <type\_na\_delegate> <kakwi\_argumenti\_ima>;

lambda func :

(type var) => {}

var => (expression)

Анонимни типове var = new {.....} позволява да правим класове без да правим класове