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:

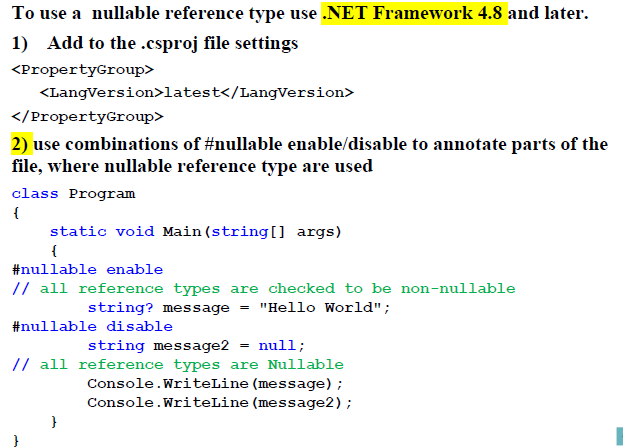
Трябва да пишем пред класовете public.

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.

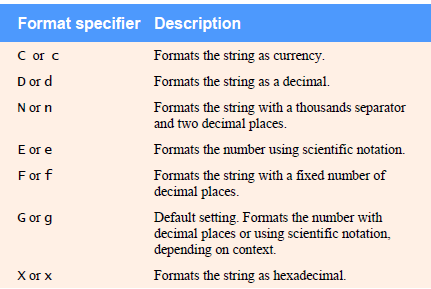


Tuples - когато искаме да пакетираме няколко неща в едно нещо. Например искаме функцията ни да върне наредена твойка инд стринг дабъл -> ползване този тип (int, string double), като пред всяко можем да слагаме имена и след това когато ползваме променливата, която съдържа тези данни да извикваме желаните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

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

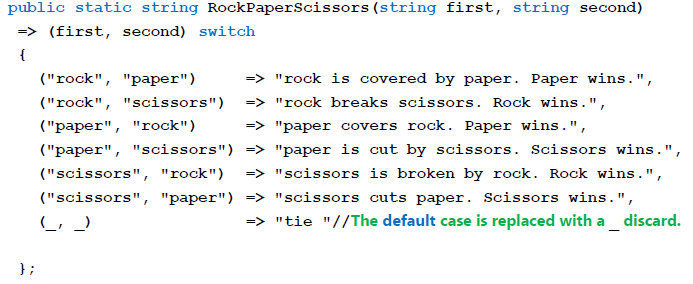
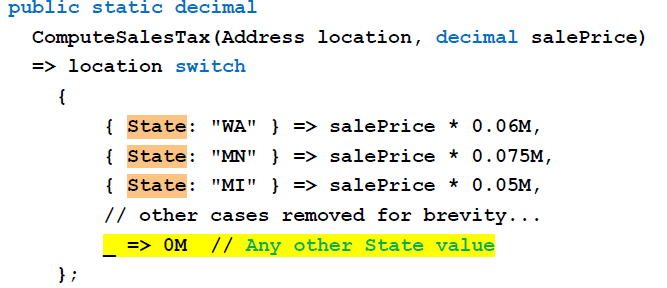
(int dummy, int max, int min) MinMax(int i)

=> (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

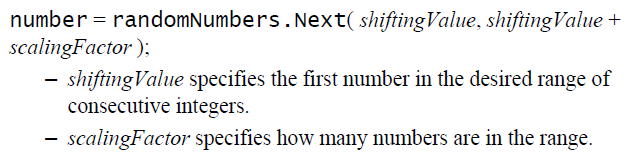
Полезно:

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:

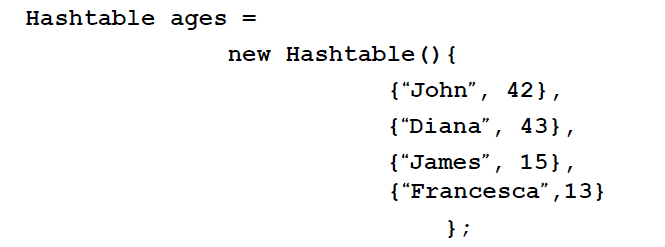
ArrayList - Remove(само пренарежда елементите),Add(ако се наложи се преоразмерява), Insert(ако се наложи се преоразмерява), [] (но не е препоръчителен заради лоша производителност, ползва се за хетерогенни колекции) вместо това->

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}

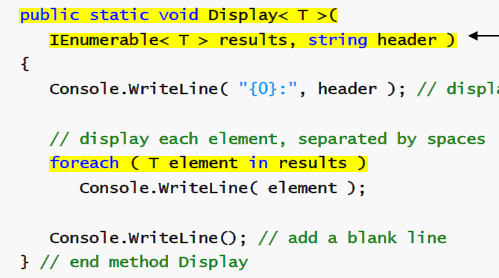
3) избираме select x (избери тези x)

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

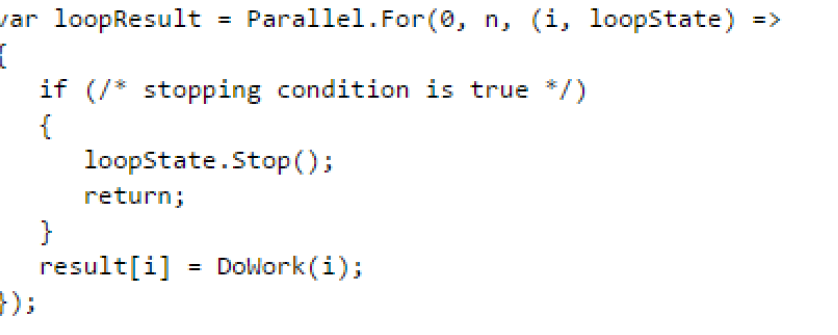
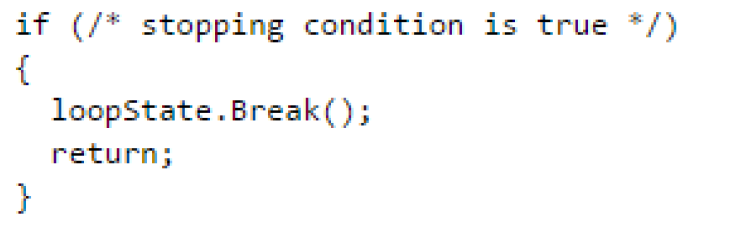
TPL: Task parallel library

**System.Threading.Tasks.Parallel**

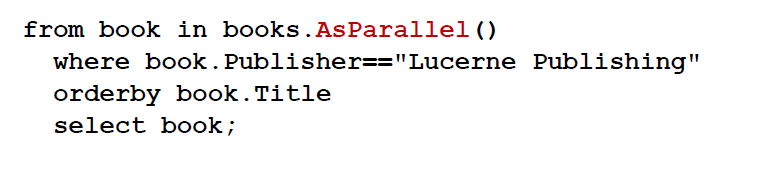
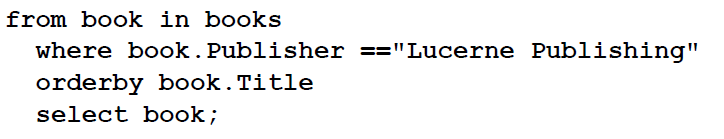
Parralel.For/Foreach

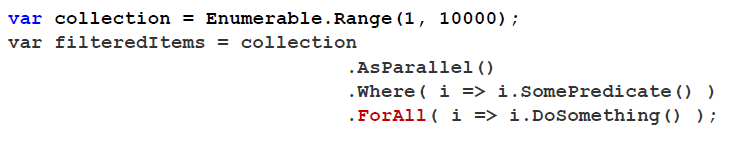
Parralel.Foreach(srcColl, itm =>Process(item)) - операциите в ламбда израза трябва да бъдат независими

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

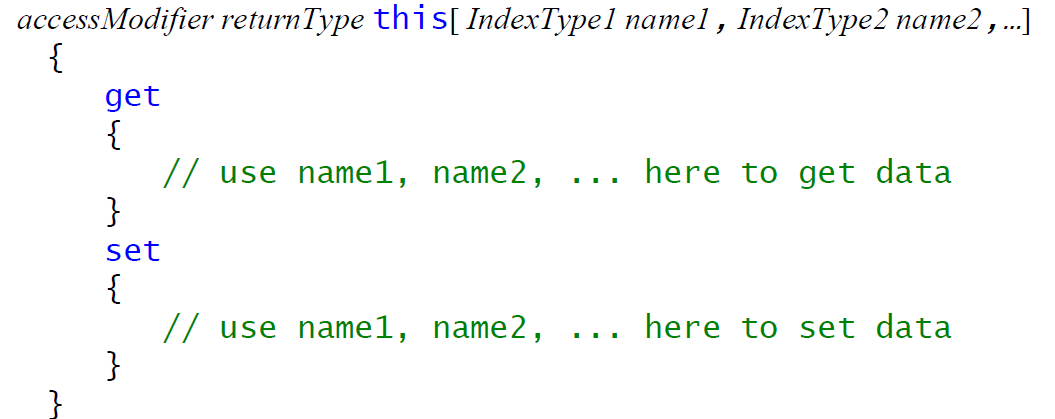
yield - яко нещо

Седмица 6

Полезни неща:

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

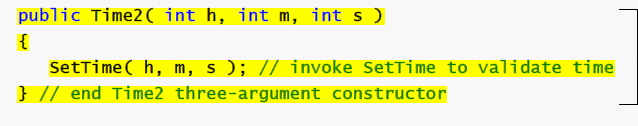
индексъри indexers - класове който инкапсулират списъци с данни може да използват this за да дефинират индексъри, които дани позволят да имаме нещо подобно на масивен-достъп до тези елементи на класас. Индексърите могат да връщат всякакъв тип. Изисвкат this за да се дефинират.

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.

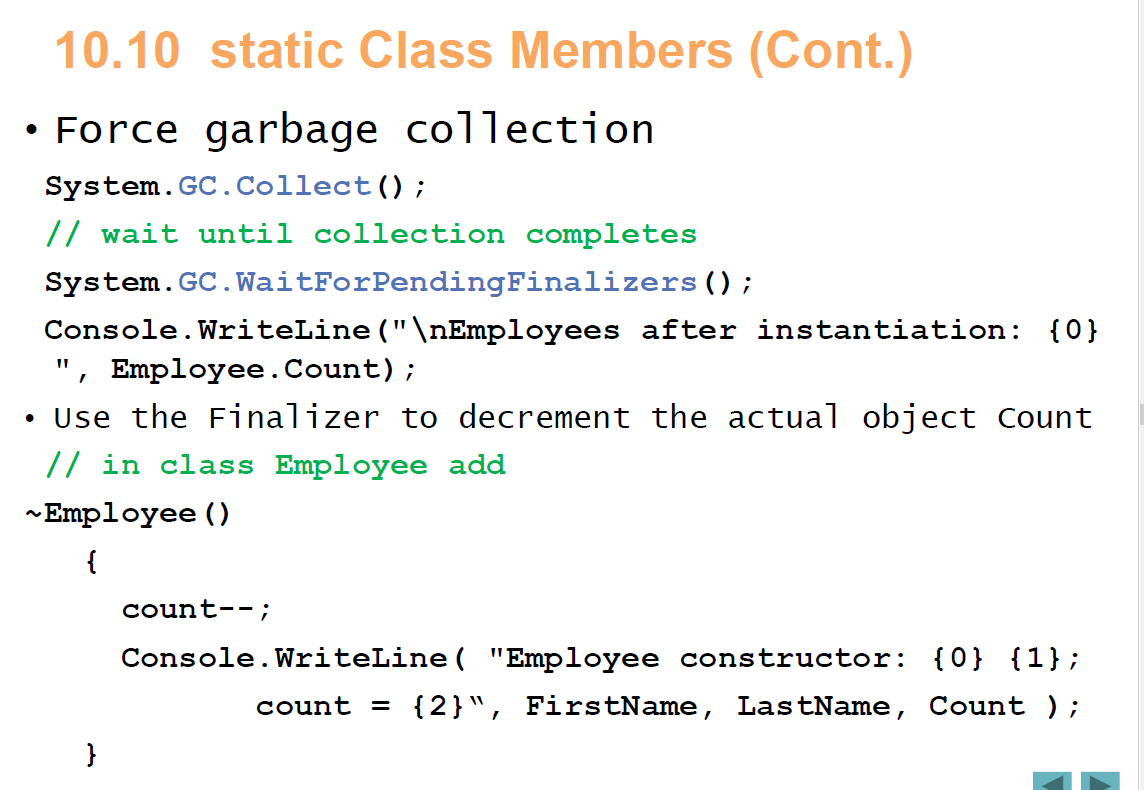
4) Interface segregation principle - не трябва клиентите да имплементират интерфейс или да бъдат принудени даползват методи които не позлват. Класа не трябва да има интерфейс, ако семантиката му не поддържа функционалността му.

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:

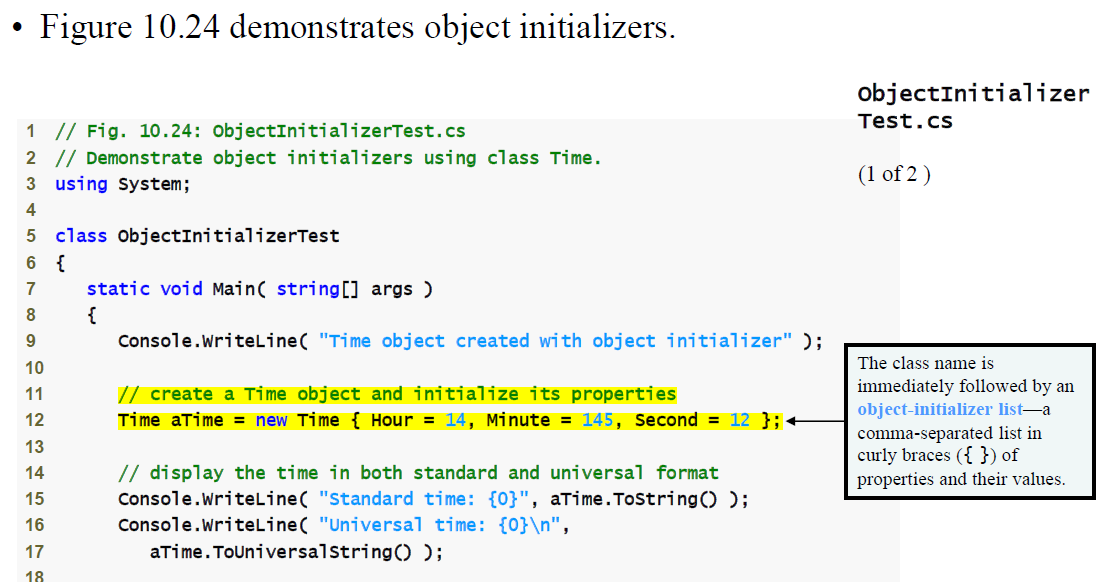
1) всички класове вътре трябва да бъдат публични

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}

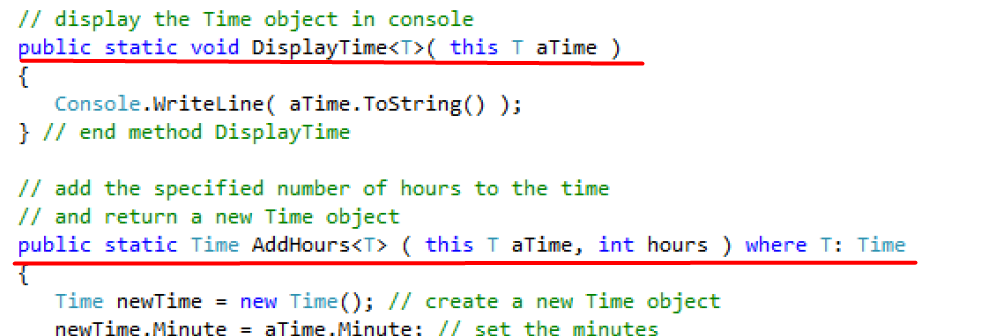
разширяване на клас с функция:

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

Седмица 7

Полезни неща:

Наследяване (IS-A)

базовия клас представя голям брой обекти

наследниците са по специализирани класове на тези обекти

protected members:

- от базовия и наследниците

- base. достъп до базовия клас

Може да променяме данни който са прайвит в базовия клас само чрез методи които не са прайвит и такива пропъртита от базовия клас.

Всеки клас наследява обджект класа

override - променяме метод на базов клас и трябва да е деклариран virtual

ПЪРВОТО НЕЩО КОЕТО НАСЛЕДНИЦИТЕ ПРАВЯТ В КОНСТРУКТОРИТЕ СИ Е ДА ИЗВИКАТ БАЗОВИЯ КОНСТРУКТОР НА КЛАСА КОЙТО НАСЛЕДЯВАТ

class name : object (става и имплицитно ако не го опоменем)

не се overrideва метод като му сложим друг модификато за достъп

:base - в конструктора викаме конструктора на бзаовия клас на наследения клас

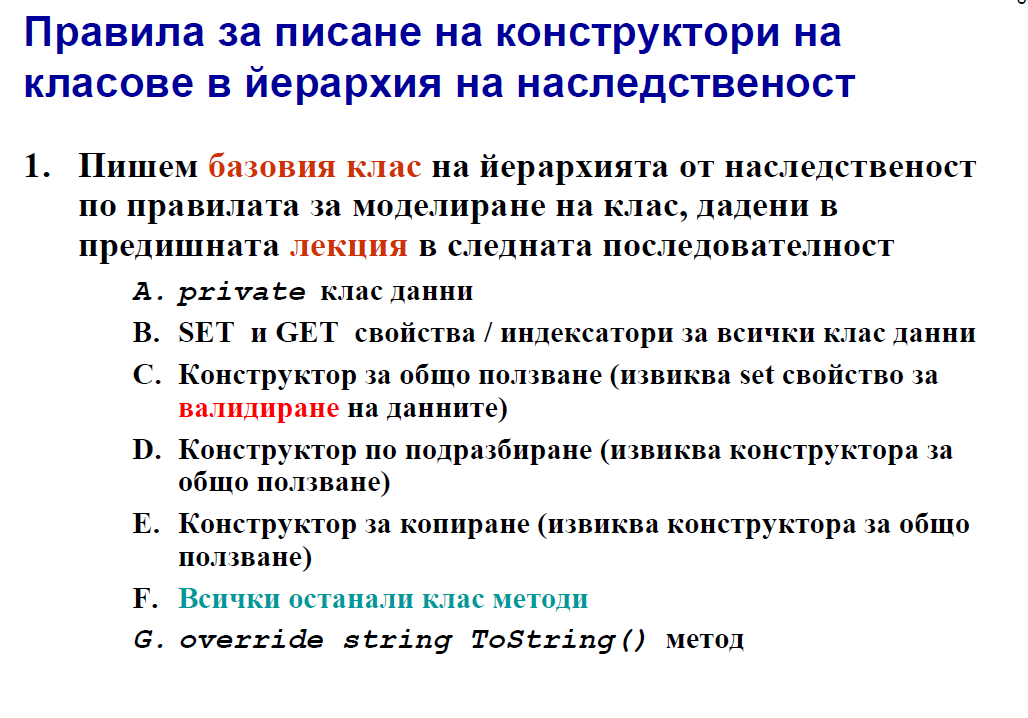
в базовия клас може да правим променливите protected, ако те ще се ползват в класа на наследниците на този клас, също и някой функции мога да са се правят със следната структура - public virtual type name(); за да могат да се пренаписват в наследниците

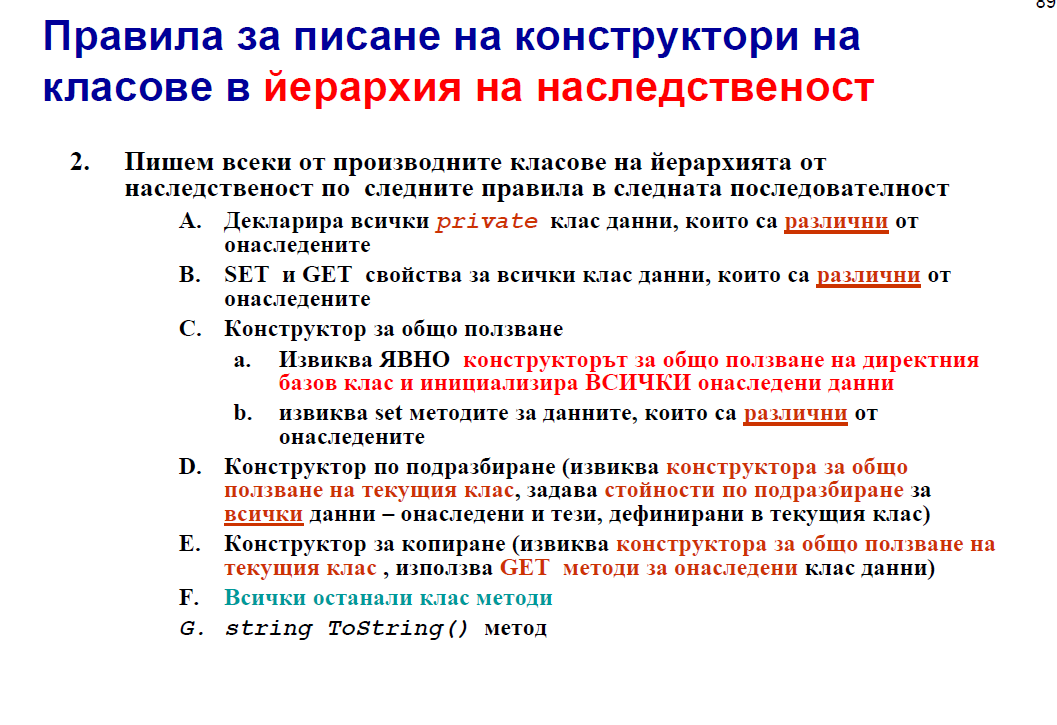
-минус на прайвит данните е че могат да се подават нвалидни стойности, наследниците пряко зависят от имплементацията на базовия клас. чуплив код

Добра практика:

- всички данни са прайвит

-публични гет и сет



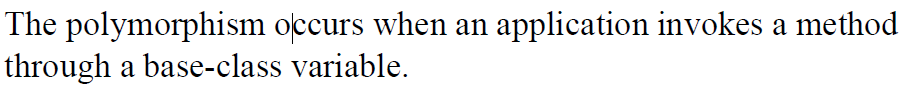


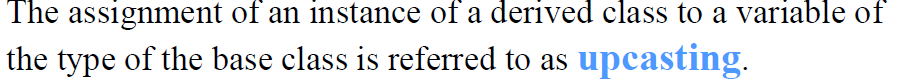
Седмица 8 - полиморфизъм, интерфейси, оврълодуване на оператори

Полезни неща:

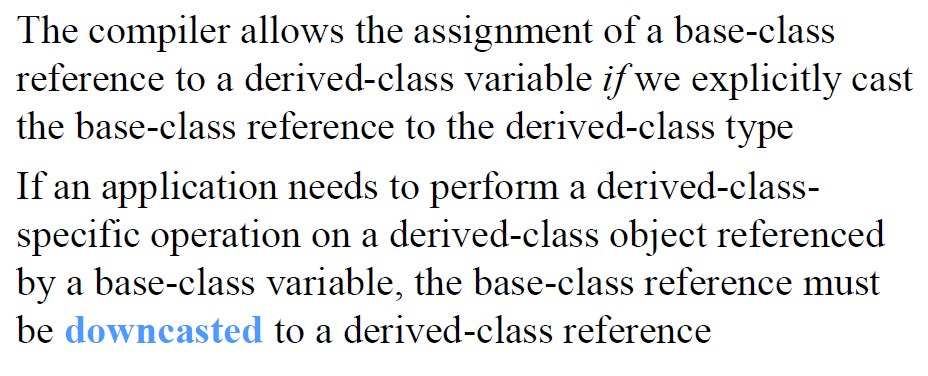
**Polymorphism**enables you to write applications that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.

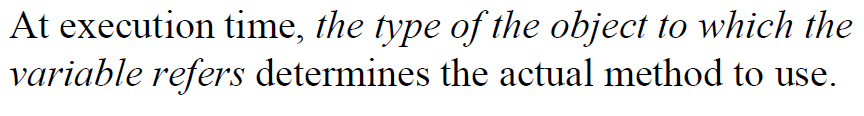
Polymorphism can improve extensibility.





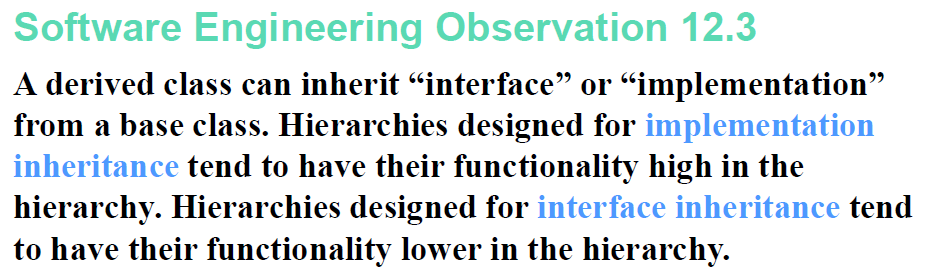
**Polymorphism promotes extensibility**





абстрактни класове или базови такива не могат да създават обекти, те обозначават само какво е общото между класовете които ги наследяват, **имат поне един абстрактен метод/свойство, който започва с думата abstract,** и в тези класове няма имплементация. **няма конструктори, няма и статични методи, декларират се само общи атрибути и поведение, които наледяват от тях директно/индиректно, наследниците трябва да овъррайднат нещата от абстрактния клас.**

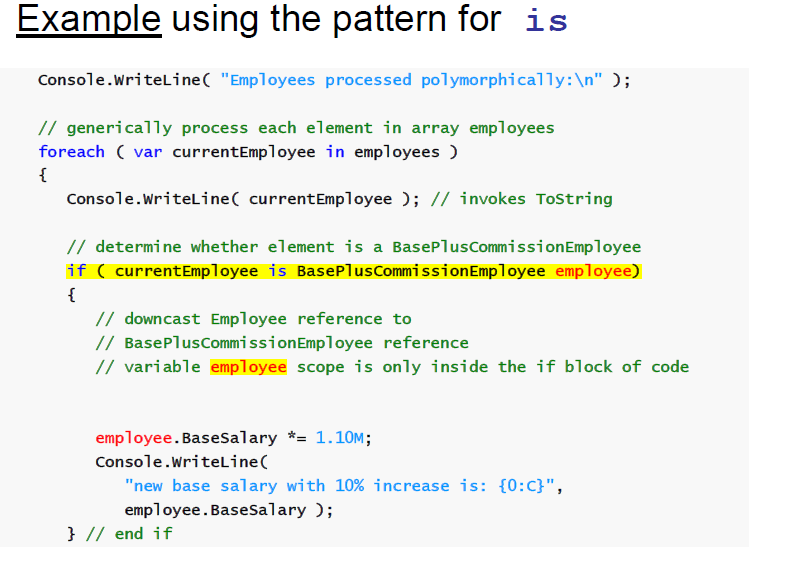
работа с абстрактни методи: имплицитно те са вирутални методи, **override** може да пренаписва абстрактен метов в наследник клас, а **абстрактите методи** мога да пренаписват базов клас с думичката **виртуал, и override.**

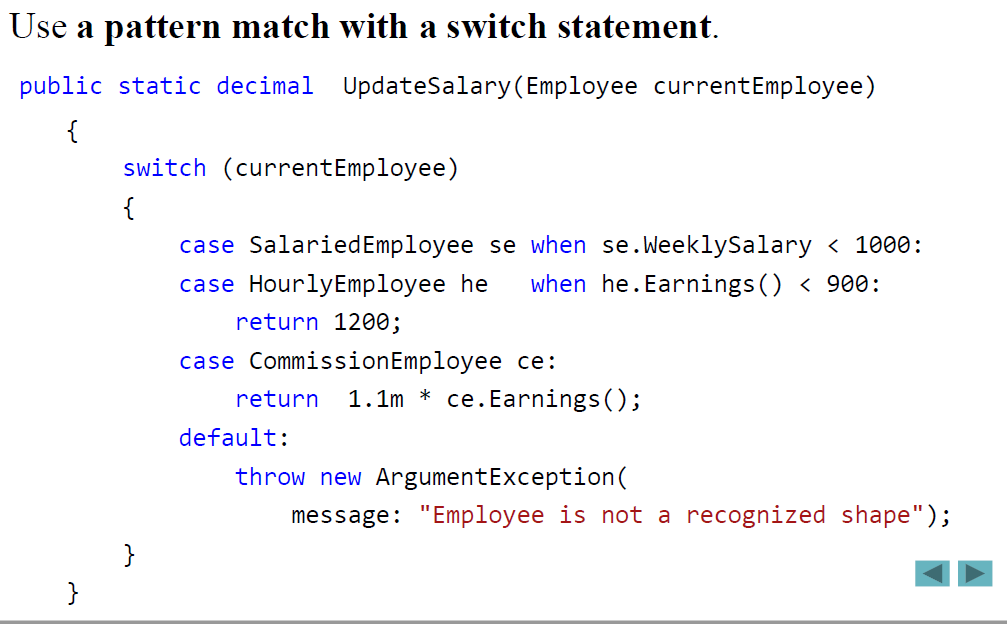


**is (varName is <TYPE>) true/false**

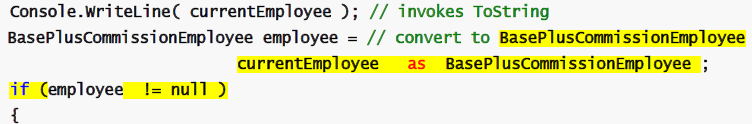
**можем да правим масив от базовия клас и в него да слагаме обекти на производнитен а този клас, и ги ползваме като ги кастнем към съответния им тип арр[1] -> тип чрез is**

**GetType() - връща ни типа на нещото**

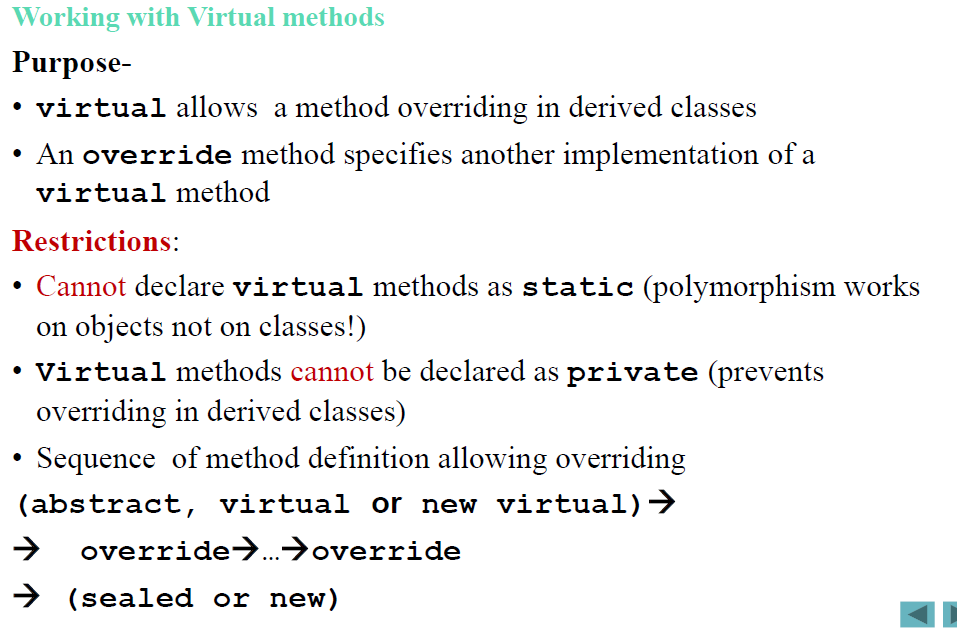


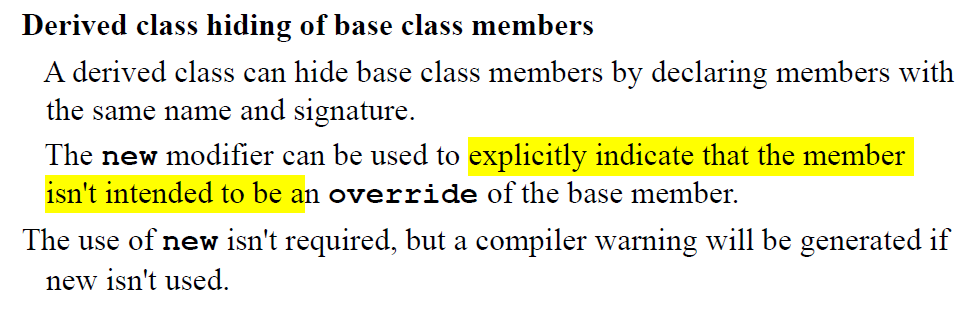


оператор **as** - за да се избегне InvalidCastException ако кастването е невалидно, изразът ще бъде null,



**sealed** - за методи които са публични да не могат да бъдат пренаписвани, прайвит методите са такива имплицитно, стаичните също...за класове, тези класове не могат да бъдат базови





Интерфейси:

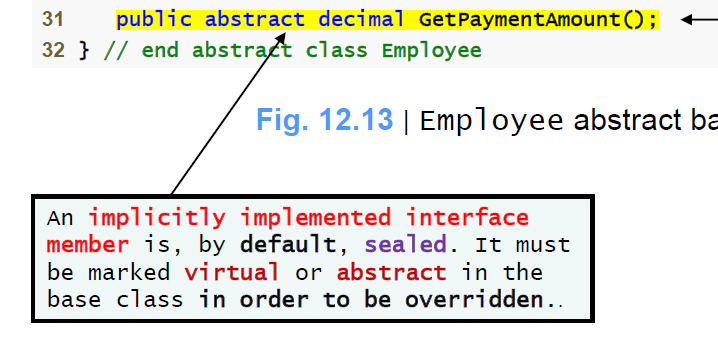
- набор от методи, които могат да бъдат викани върху обекти, все едно да върнем някаква информация от нещо, дефинират и стандартизират начините по които хората и системите си взаимодействат помежду си:

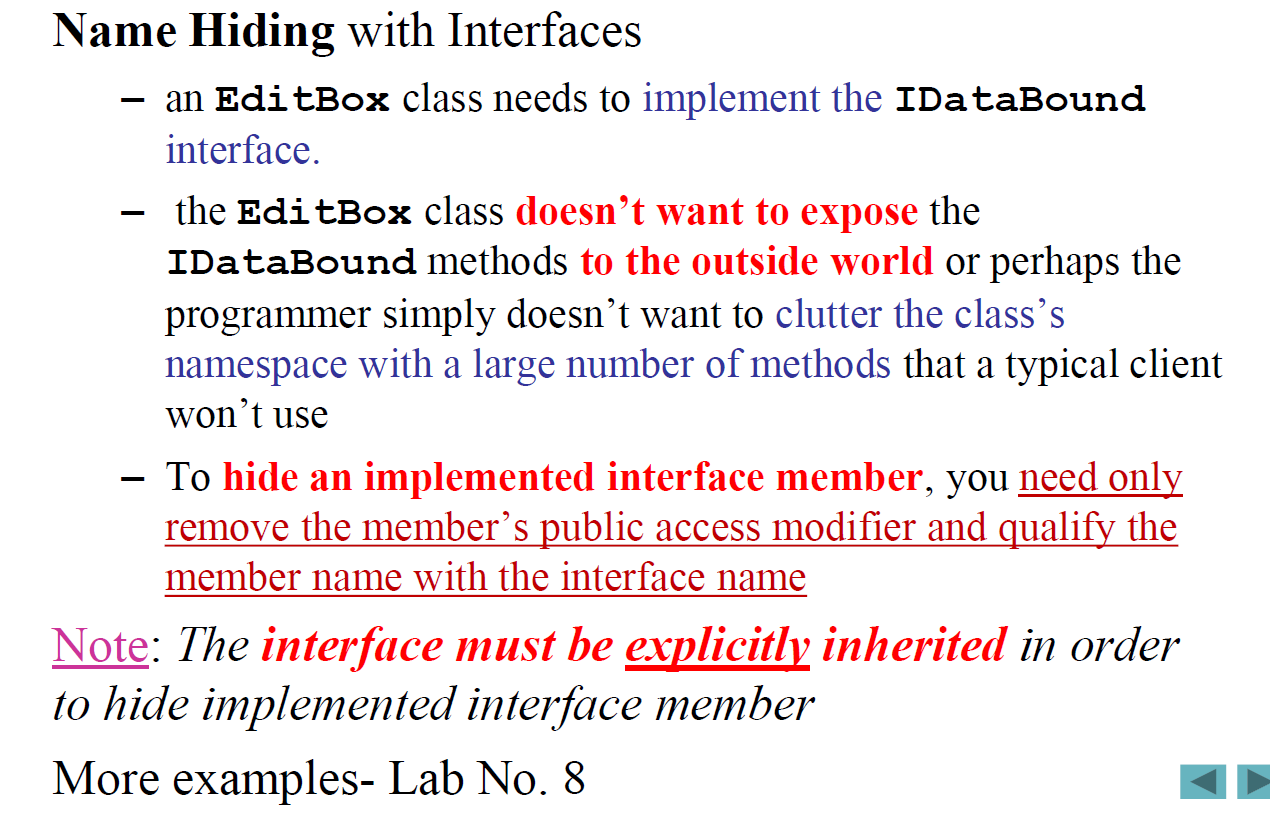
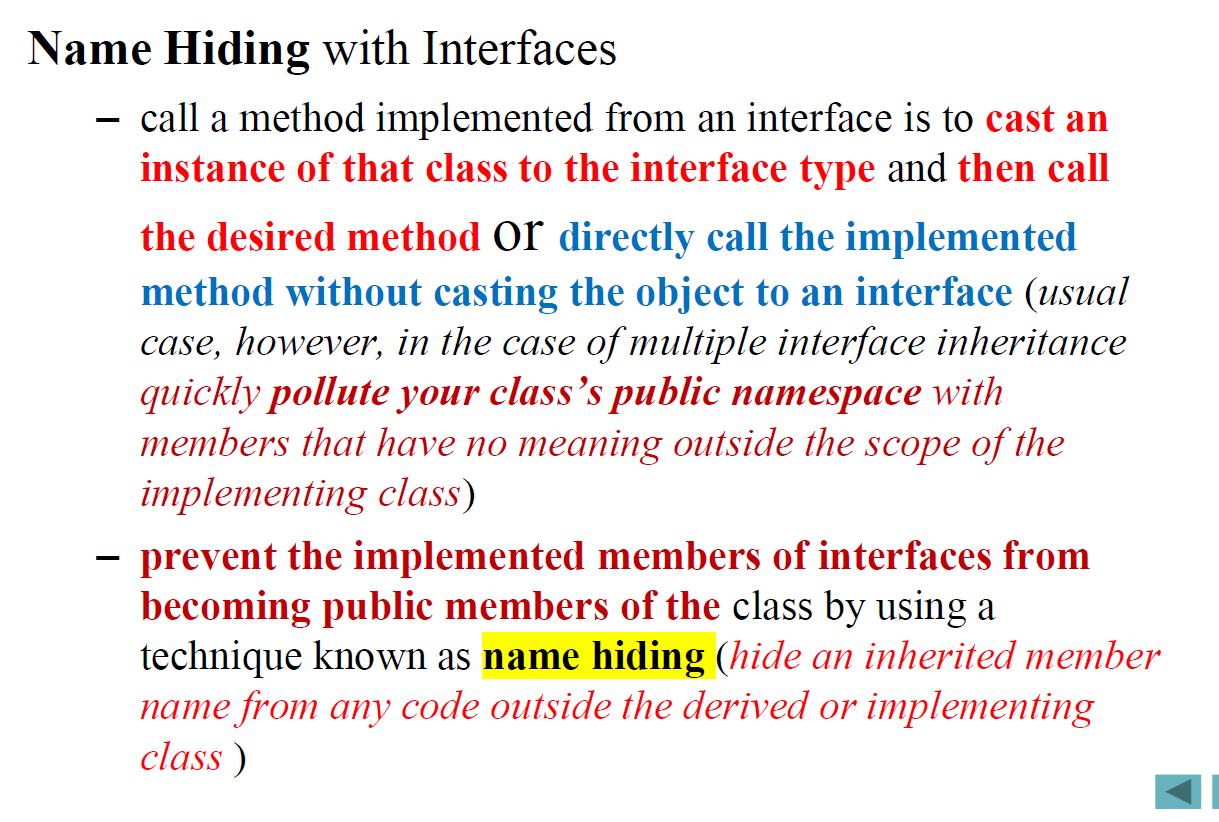
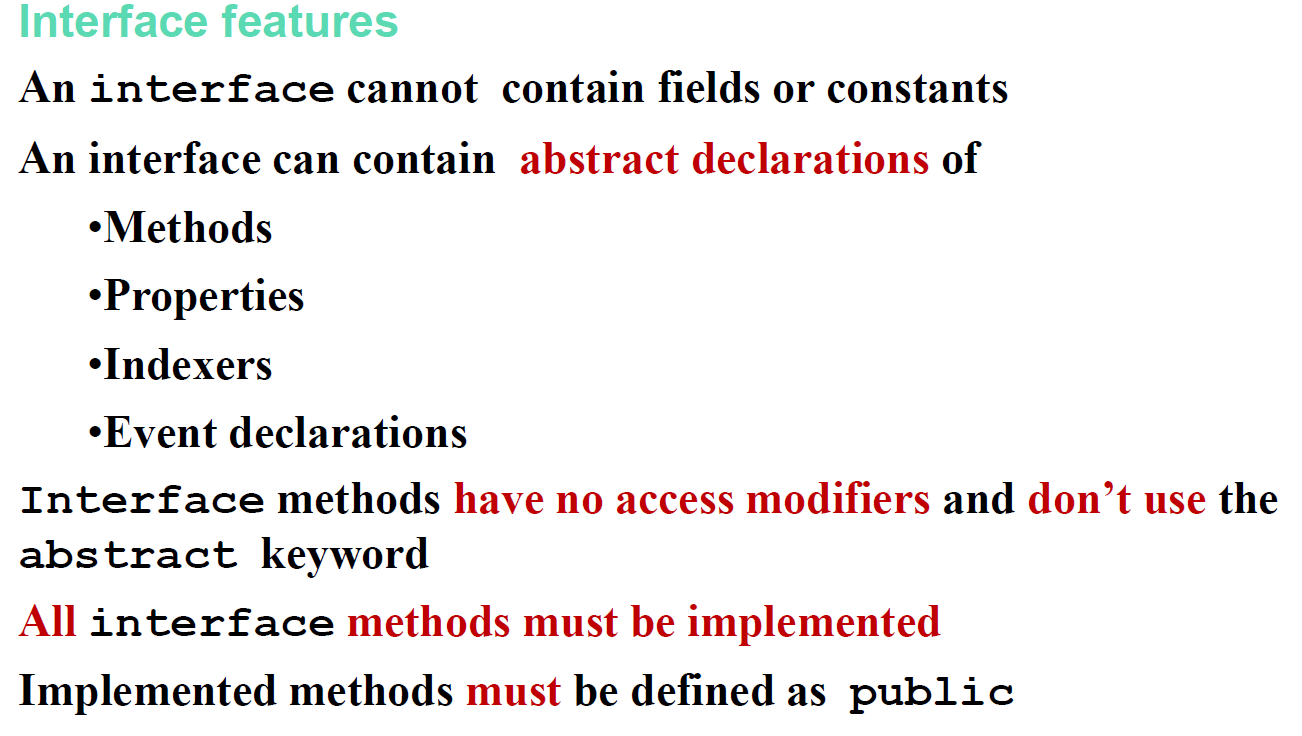
**interface - може да има в себе си абстрактни методи, пропъртита индексъри и събития**

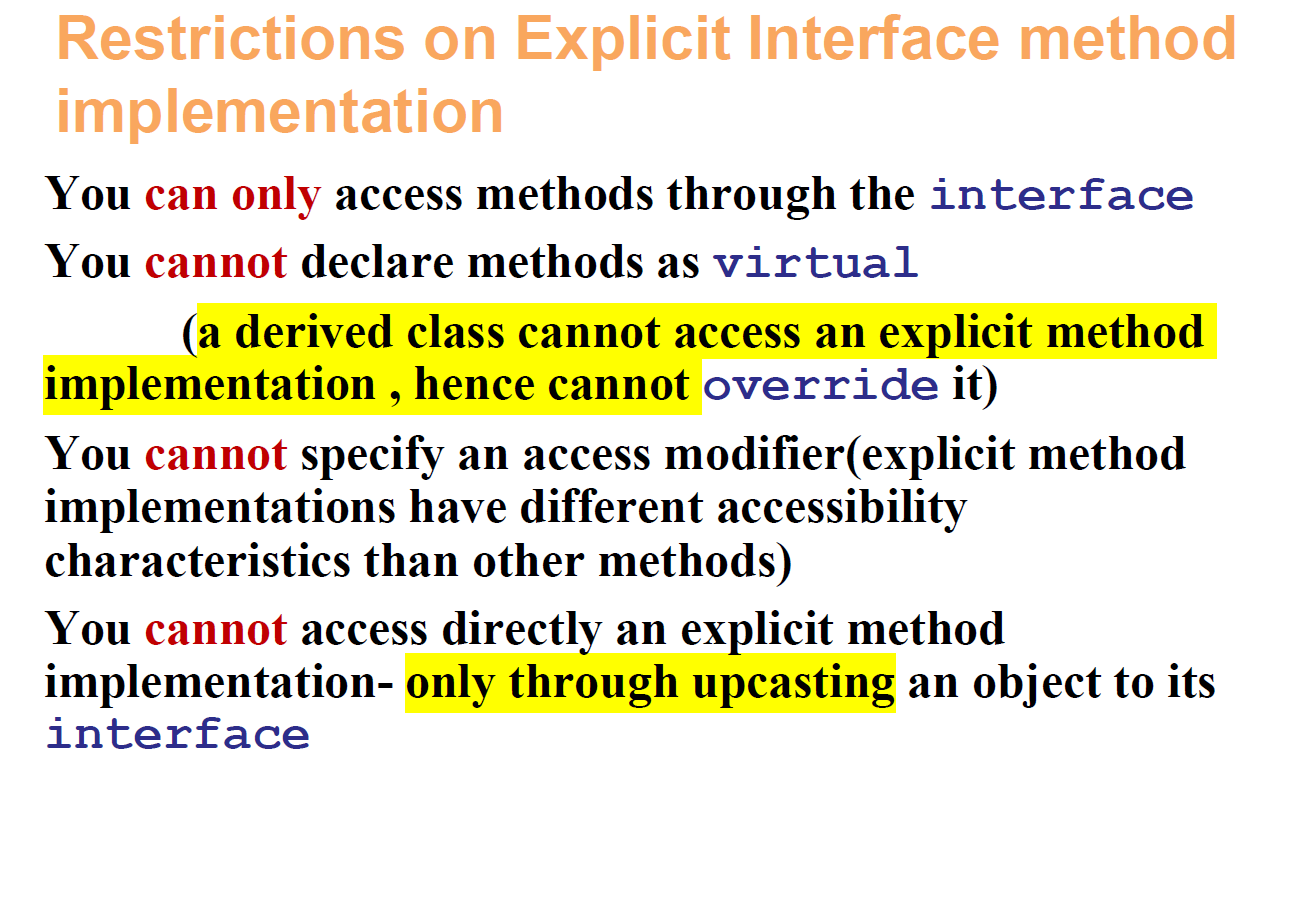
**не са производни на object класа, не е конвертируема до обджект класа, тя е конкретен тип,interface предоставя достъп в рънтайм към всички методи на клас обект + тези в интерфейса.**

**ползваме - клас трябва да упомене че имплементира интерфейс като листе с : в декларацията си на класа. конкрете клас трябва да ползва сигнатурата на интерфейса като имплементира функции от него иначе класа е абстрактен, трябва да се упомене като такъв и да съдържа абстрактна декл за всеки неимплем член**

**ИМЕНАТА НА ИНТРЕФЕЙСИТЕ ПОЧВАТ С I, и методите вътре трябва да са с имена на попринцип вършещи се неща от общ характер, понеже ще се правят за няколко различни нерелациращи класове. то тогава се ползва интерфейс като няма как тези класове да наследяват едно и сщо нещо.......ТЕ СА ПУБЛИЧНИ и абстрктни**







**когато искаме да ползаме дадена функция от конкретен клас, трябва да ъпкастнем до съответния интерфейс или да проверим, когаот имаме повтарящи се имена от два интерфейса да порверим с оператора “as”**

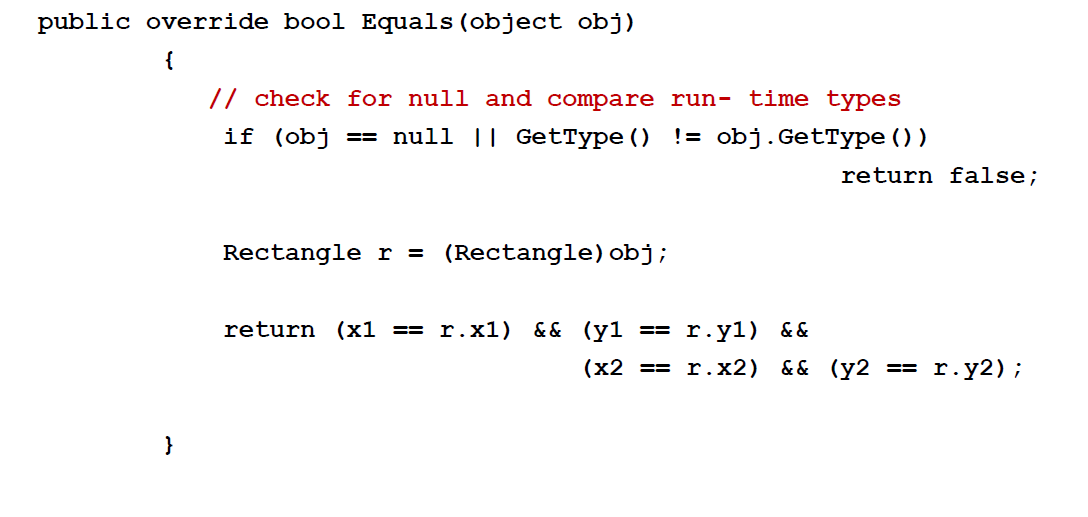
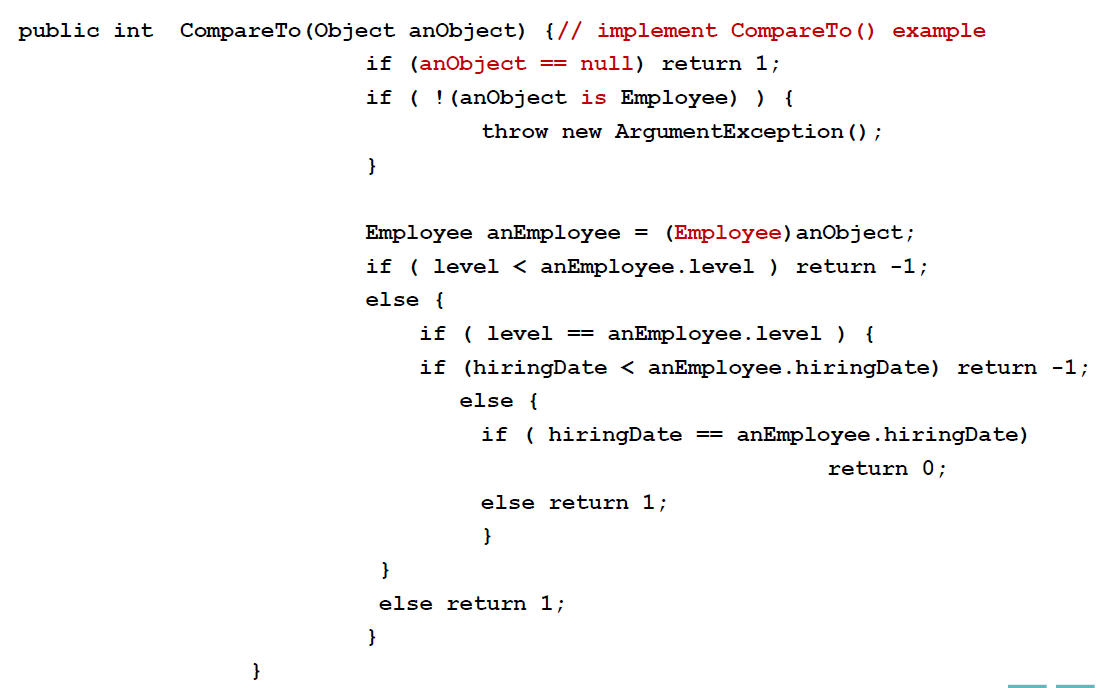
**overloading на operator - трябва да са публични и статични и да са в клас и връщат нов обект**

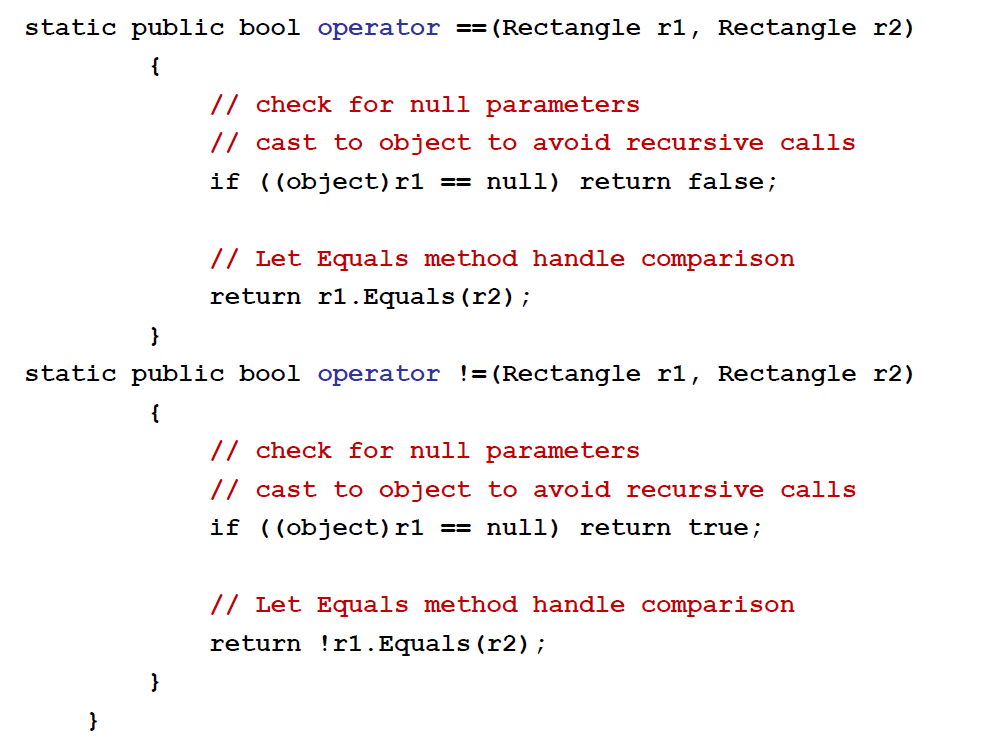
**public static type operator ‘symbol’(first-left, second right)**

**Equals() + GetHashCode()**

**== !=**

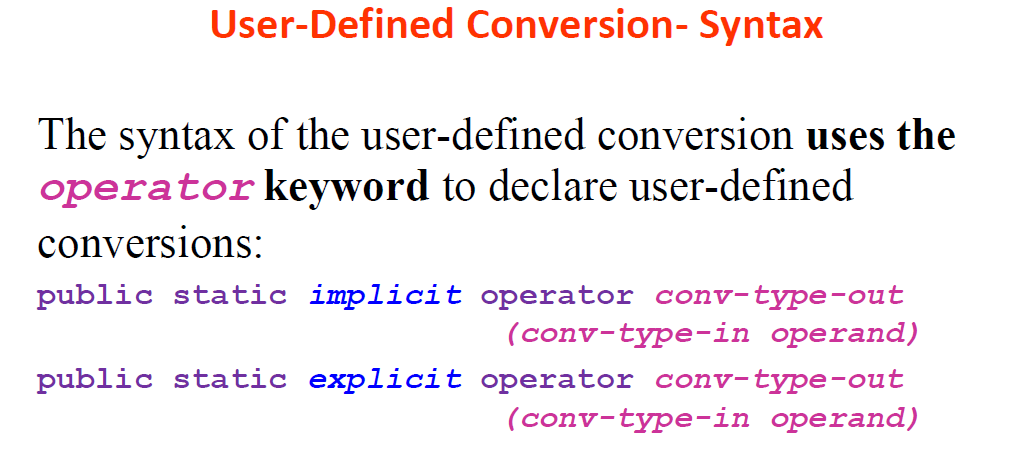
**IComparable**





**първо проверяваме дали не е нул или от различен тип, после ако мине това кастваме към нашия си тип**

каствания:



трябва да са статични, имплицитно ще стане автоматично, експлицитно ще стане като се окаже - начи explicit operator (myType) (args) {new obj myType return (obj)}

implicit operator <nestho> (myType obj) {return (smth)}

за да ни е по лесно кастването, можем да направим един клас, който да бъде наследявант от нещата които ще кастваме, така те ще съдържат еднакви данни и ще става по чисто и лесно

Структури: те са като стойностни типове, подавта се по стойност не по референция, имат само конструктор за всеобщо ползване, нямат инишълайзери

Nullable type -> <type>? myType ...

Седмица 9

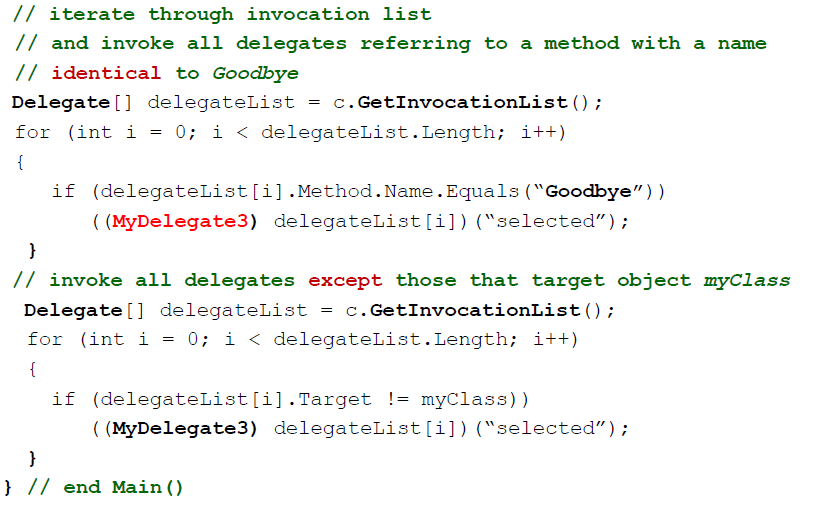
Полезни неща:

Делегати

обект, който съдържа в себе си подреден списък от методи с еднаква сигнатура и тип на връщане, списъка с методи се казва **invoecation list** когато делегат е invoked той вика всеки метод от този списък, **singlecast, multicast**,

мултикаст делегати:

static methods, Combine/Remove...Delegate[] GetInvocationList...Target/Method



operator+= - създава нов списък от делегати + този който се подава

оператор-= - премахва указания елемент(края към началото)

като извикаме делегат с няколко функции без да връщат стойност той ги изпълнява на един път с подадения аргумент, **важно е като го викаме да проверяваме дали не е NULL**

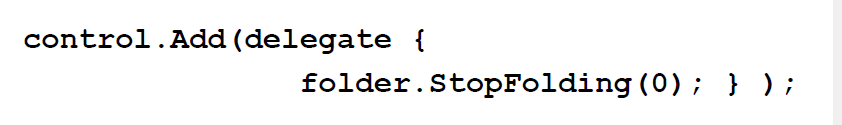
**ако делегатът има методи коит овръщат стойност получаваме стойността от последния метод а другите се игнорират**

**Защо ги ползваме:**

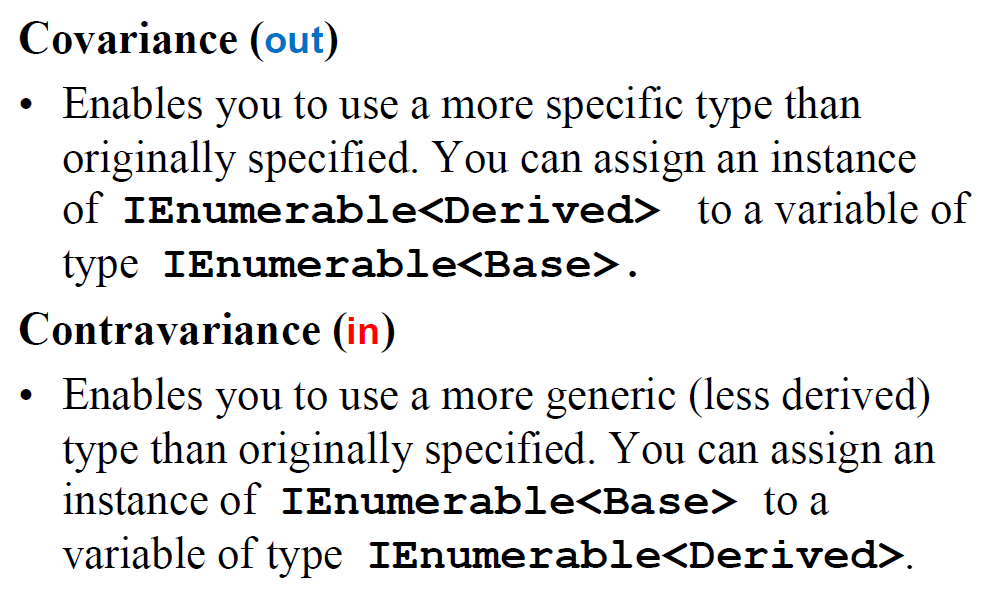
**обратно извикване, асинхронни събития,**

**Анонимни методи - отдясно при добавяне в делегат, инициализатор при деклариране на делегат**

Adapter method - примерно имаме група от методи които приемат 0 аргументи, и имаме един метод с Х аргумента, правим адаптор метод без параметри и вътре в себе си той вика функцията обаче с някакви параметри. / или да използвме анонимна функция която да прави това (тук няма нужда да създаваме нови функции нови имена)



contravariance in / covarianve out - out когато ще се ползва само като връщащ тип, in когато ще се ползва като параметър в интерфейса или делегата



Стандартни делегати:

**Action, Func, Predicate**

**public delegate void Action<in T>(T obj) T- този тип или тези които са по малко наследнени**

**public delegate TResult Func<inT, outResult>(T arg )**

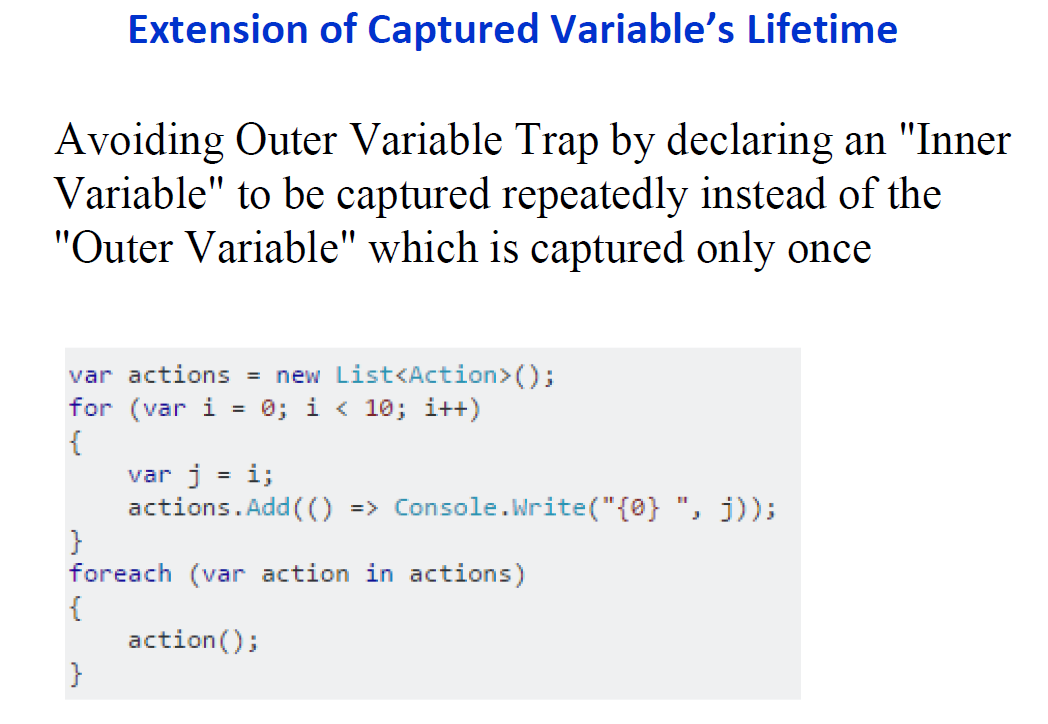
**?? нулл coalescing operator**

**nameof(value)**

**public delegate bool Predicate<inT>(T obj)**

**lambda methods**

**(type a, ... type b) => {body} , () => {...} zero arguments**

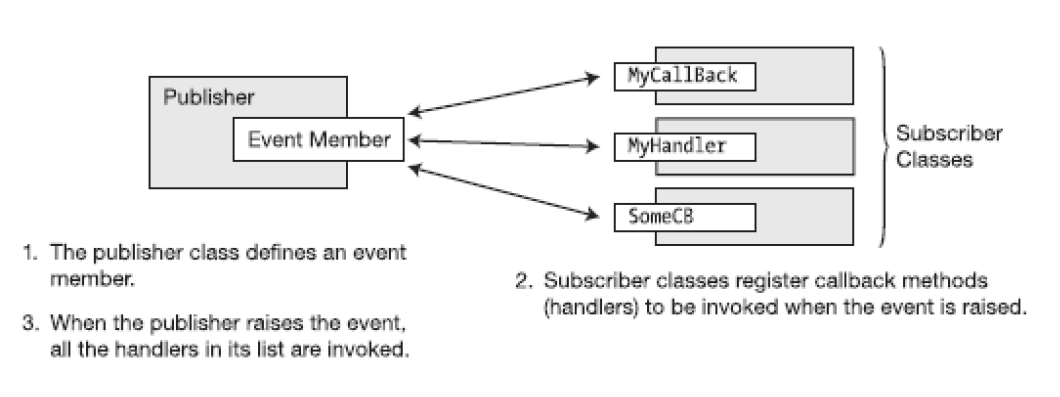


**Event Handling**

**Събитие е нещо което се изпълнява когато някакво условие е изпълнено.**

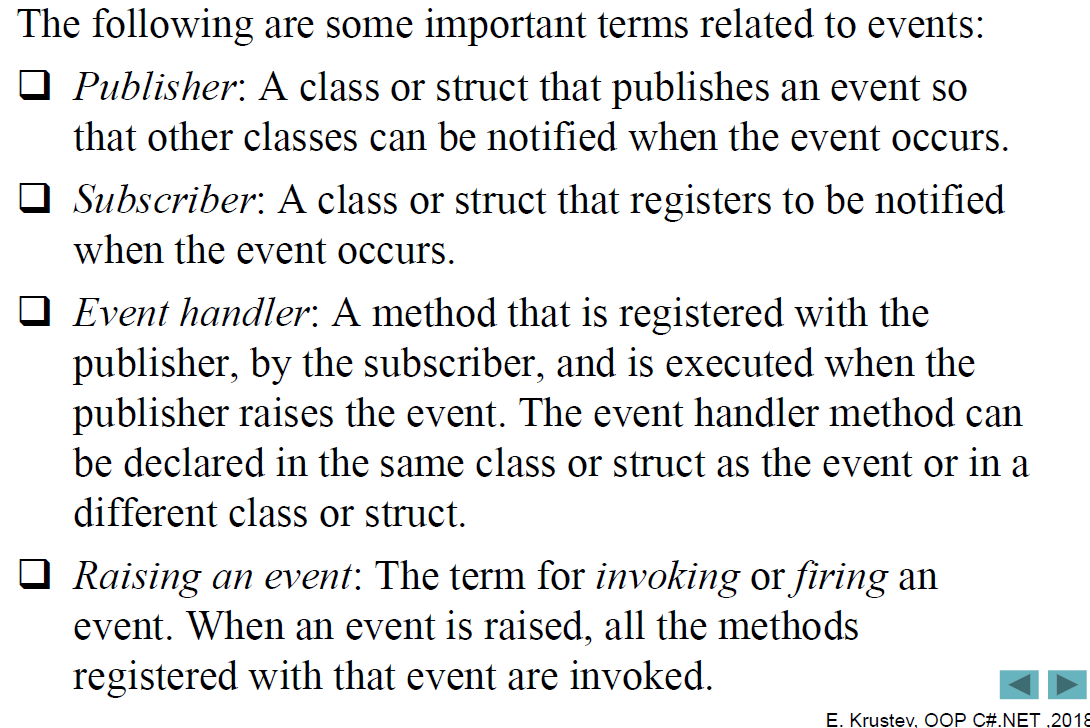
**Publisher/Subsriber pattern.**

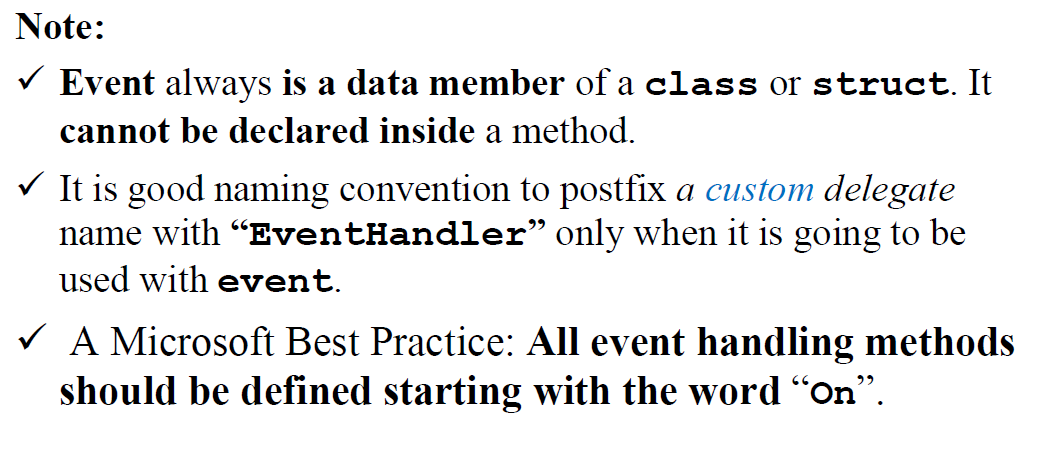
**publisher (event source) - set of events**

**subsctiber(event consumers) - sign up for notification by supplying a method EVENTHANDLER METHOD** 

**event delegate\_type On<eventname>**

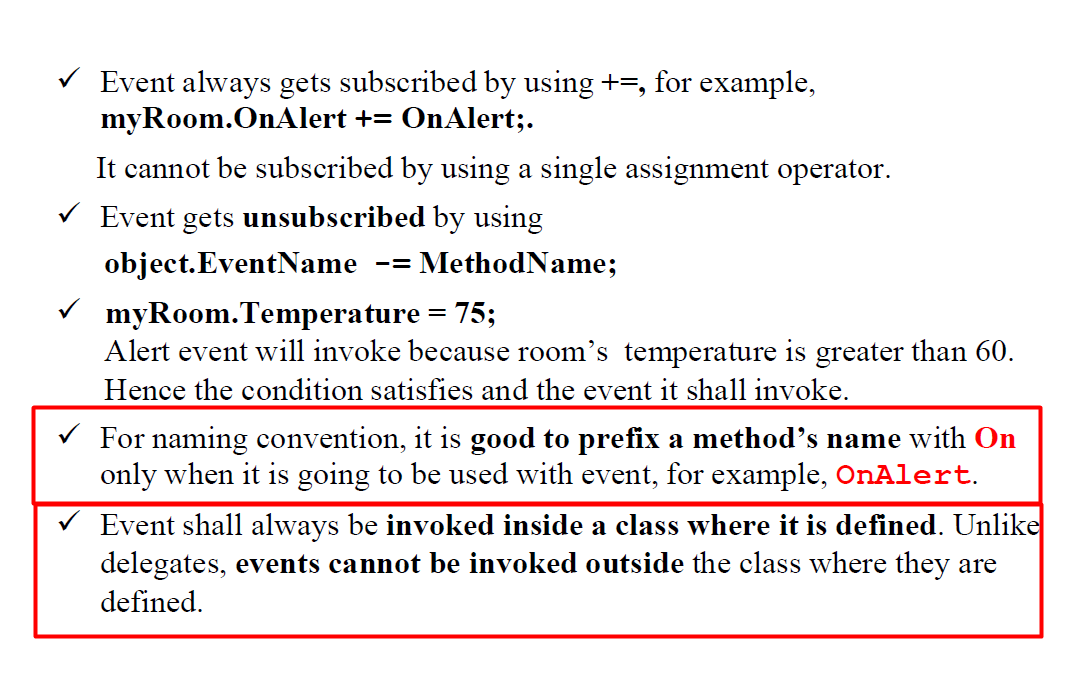
**event в интерфейс -> delegate void Event ,,,,,, class Person{ public event Event eventname}**



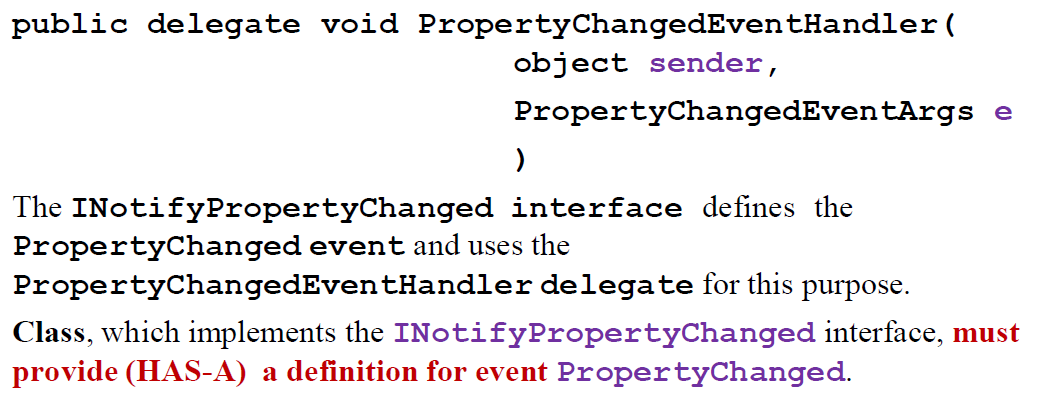


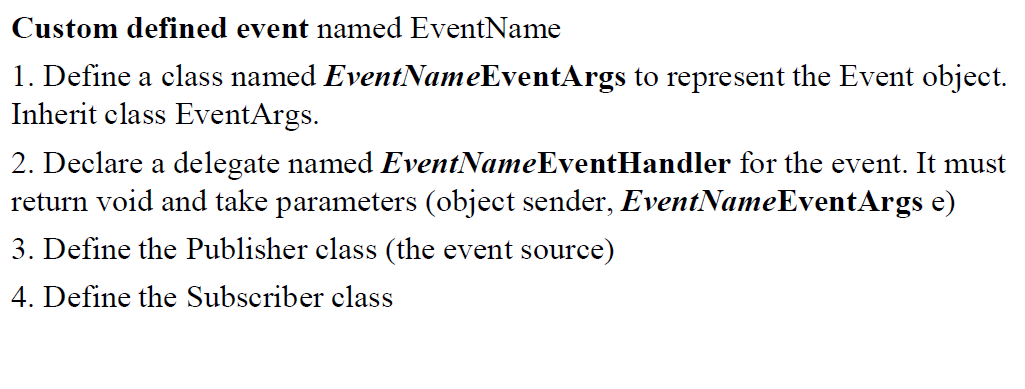
**public delegate void EventHandler(object sender,EventArgs e);**

**EventHandler - за воид тип, първи параметръ е този който реферира инстанцията, където евента е бил деклариран и изтрелва събитието, втория EventArgs държи данни за събитието, ако няма данни за пренасяне ще е празен иначе ще е капсолован в тип EventArgs.**



**PropertyChangedEventHandler - ползва се когато пропърти се промени**





Седмица 10 - GUI with WPF

Полезни неща:

How XML namespaces help provide unique XML element and attribute names.

To customize the look-and-feel of WPF GUIs using styles and control templates.

To use data binding to display data in WPF controls.

Елементи - започват и завършват с таг

Атрибути - са от вида нещо-си = “нещо-си”

**Border, GroupBox, Window, Expander - групират елементи**

**Canvas, DockPanel, Grid, StackPanel, WrapPanel - панели**

**- HorizontalAlignment, VerticalAlignment, Margin**

**- horizontal - left center right stretch**

**- vertical - top center bottom stretch**

**Button - Visibility(hidden, collapsed), Vertical/HorizontalContentAlignment**

**StackPanel - Orientation(Horizontal, Vertical), Flow Direction(Righttoleft)**

**Grid - ShowGridLines=True, row,col почват от 0, Grid.ColumnSpan, RowSpan**

Binding: with ALT + Letter



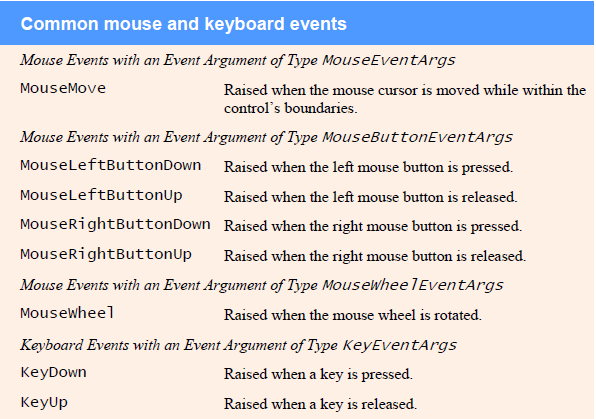
Button - Click

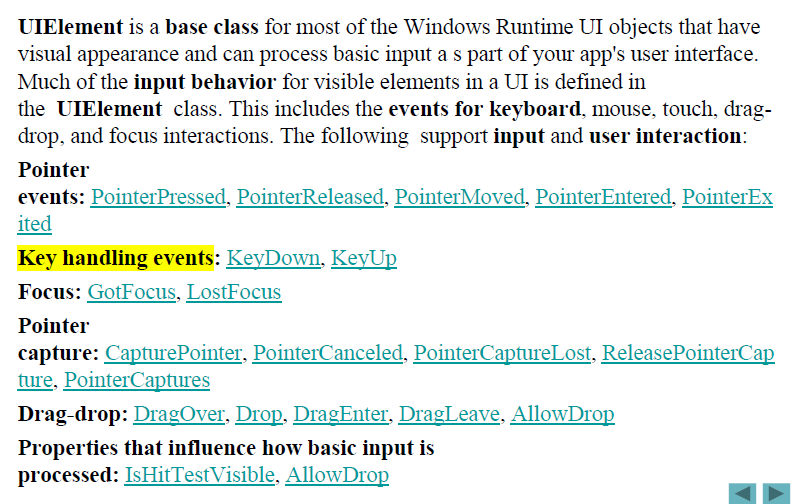
Radio Buttons - GroupName

ListBox - Освен стрингове, може и елементи - Button, TextBlock, SelectionChanged





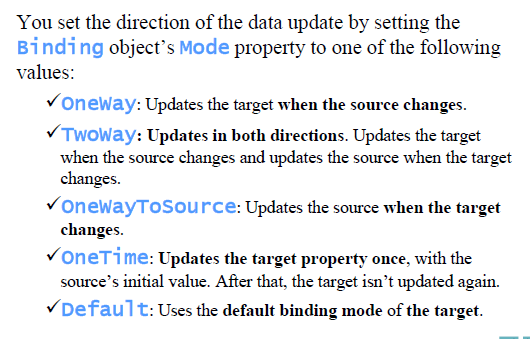




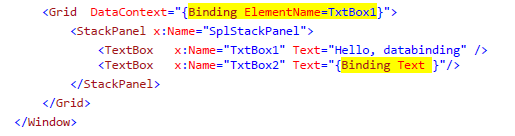
Data Binding:

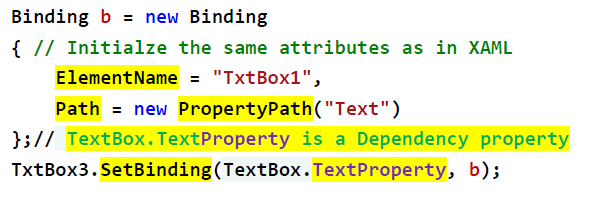
Как става, правим един таг който ни е сорса, и му даваме някакво име. След това в друго нещо което има същото проперти като това към което се свързваме с таргета ни като на порпъртито което съвпада правим <пропърти>=”{Binding Path=По какво да стане ElementName=името на сорса ни}}”

има модове:



По този начин можем да зададен дефолтен дата контекст и да не пишем всеки път





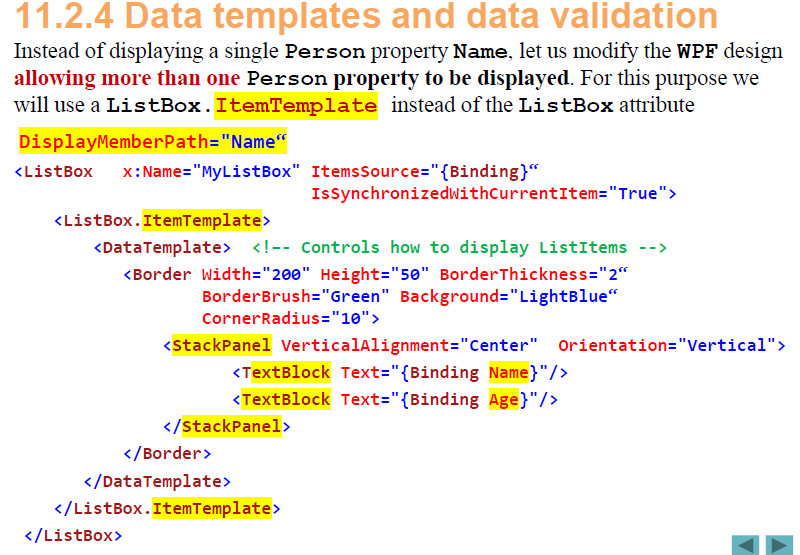
Kак да навържем листбокс с някакъв колекшън:

1) Правим си тип който да наследява INotifyPropertyChanged

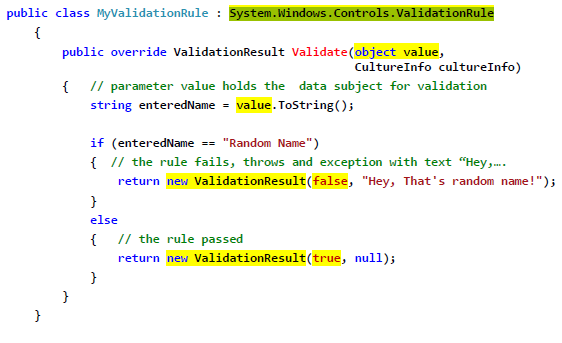
2) Правим колекция от тип ObservableCollection<Type>

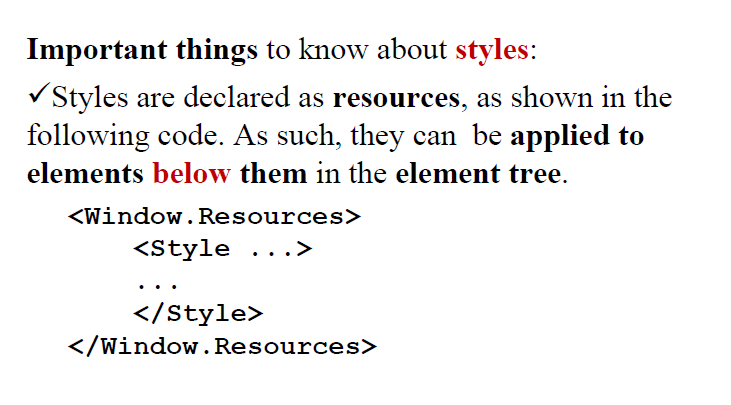
3) ЛистБокс слагаме пропъртира ItemSource={Binding} DisplayMemberPath=”Name” IsSynchronizedWithCurrentItem="True"

4) В кода насочваме на панела който ни държи лист бокса датаконтекста да е класа но се прави в кода не в ЮАЙ частта

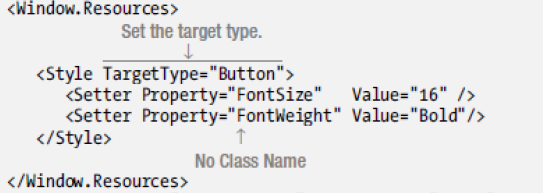


Валидиране:

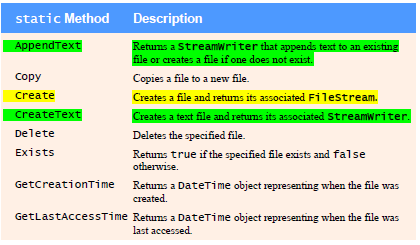
Трябва да направим клас който да наследява **System.Windows.Controls.ValidationRule** 

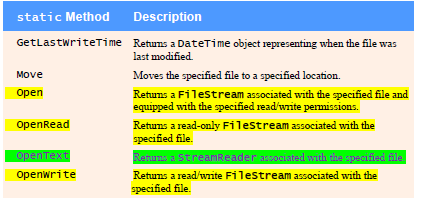


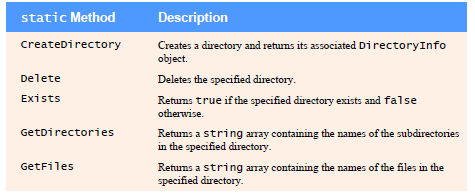


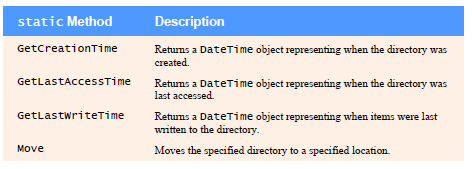


**Седмица 11 - файлове**





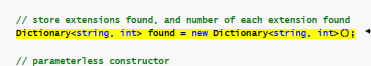


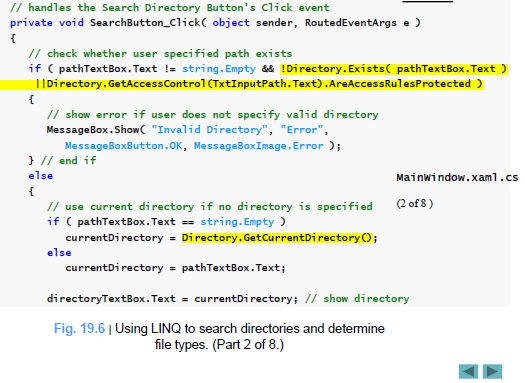


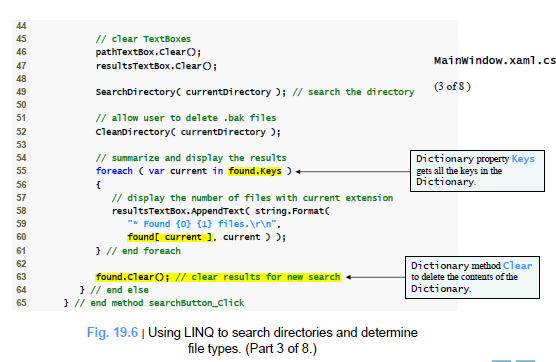
**Файлове:**

File.Exists(fileName)

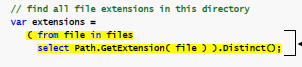
StreamReader, ReadToEnd, Directory, GetDirectories

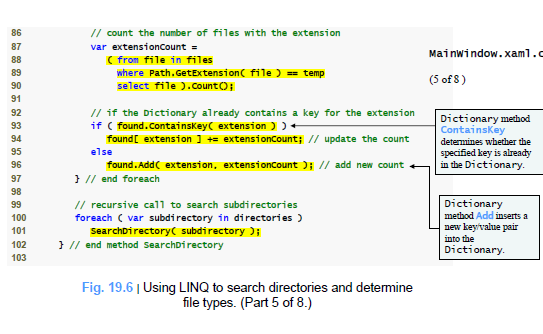






Direcotry.GetFiles(folder)





Path - GetExtensions

File - Delete

Dictionary - ConatinsKey, Add, Remove, Keys, Values, Clear

OpenFileDialog