

Упражнение №2 по ПС - Част 1

C#

Още разлики между C++ и C#.

Целта на това упражнение:

Запознаване с езика C#.

Необходими познания:

От предметите дотук и предното упражнение трябва да можете:

- Да използвате езика C++
- Да създавате променливи и функции
- Да използвате операторите if, for, switch, указатели,...
- Да създавате class, struct, interface, enumeration
- Да управлявате изключения
- Да създавате проекти с повече от един файл
- Да създавате конзолни приложения
- Да познавате общата структура на C#

Задачите в упражнението изграждат:

Университетска информационна система.

В това упражнение: разширяваме системата за влизане с потребителско име и парола

- класа LoginValidation ще прави проверка в предварително съществуващ списък от потребители

- подаваме на класа LoginValidation с делегат функция, която ще се изпълнява при неуспешна валидация

В края на упражнението:

Потребителя ще въвежда потребителско име и парола, системата ще провери дали има такъв записан потребител и ако има - ще го регистрира и ще изведе какви привилегии има.

Легенда:

Най-важното

- Схематично и накратко

Обяснение. Уточнение.

Пример: или **Задача:** ограденото трябва да изпълните за да е синхронизиран проекта ви

- Подходящо място да стартирате изпълнение за да проверите дали работи вярно.

1. Още разлики между C++ и C#

1.1. .Net характеристики

C# е разработен за използване под .Net. За това има специфики на езика, които произхождат от принципите в .Net

1.1.1. Указатели

В .Net няма указатели.

Едно от основните преимущества на .Net е *Garbage Collection* и *Safe Code*. За да работи то е необходимо във всеки момент да се знае всяка променлива за какво отговаря. GC трябва да знае от какъв тип е обекта, за да го освободи. *Safe Code* трябва да гарантира, че кодът се изпълнява само в адресното пространство на .Net. За това указатели няма.

В C# достъпа до членове на обекти винаги става с оператор `.` (точка), вместо `->` (стрелка), защото никога не се достъпват през указател към обекта.

1.1.2. Предаване на параметри

- **Структурите** (и базовите типове) винаги се предават **по стойност**.
- **Обектите** се предават **по адрес**.

Винаги, по подразбиране.

Пример: Следните две функции дават различни резултати:

Int → Базов тип → по стойност

```
static void SetRoleToZero(Int32 i)
{ i = 0; }

static void Main(string[] args)
{
    User u = new User();
    u.Role = -999;

    SetRoleToZero(u.Role);
    Console.WriteLine(u.Role);
}
```

-999

User → Клас → по адрес

```
static void SetRoleToZero(User u)
{ u.Role = 0; }

static void Main(string[] args)
{
    User u = new User();
    u.Role = -999;

    SetRoleToZero(u);
    Console.WriteLine(u.Role);
}
```

0

Задача: Добавете в класа `LoginValidation` три частни низови полета за:

- Потребителско име
- Парола

- Съобщение за грешка

Задача: Добавете в класа `LoginValidation` публичен конструктор, който приема два параметъра, с които да инициализира частните променливи за потребителско име и парола.

Ще даде грешка в Main метода. Ще я оправим след малко.

1.1.3. Предаване на параметри с `ref` и `out`

- Можем да укажем предаване по адрес на структури (и базовит типове) с `ref` или `out` параметри
(т.е. да укажем поведение, различно от това по подразбиране)
- `out` работи като `ref`, но гарантира изключение, ако методът не назначи стойност на този параметър

Допълнително:

- Обектите се предават по адрес, но самият адрес се предава по стойност
т.е. ако подменим целия обект с друг (т.е. подменим адреса с нов), промяната ще е видима само вътре в метода.
(т.е. `ref` е полезно и за обекти, не само за прости типове)

Пример: Следните функции дават различни резултати:


Редакция на обект

```
static void
SetRoleToZero(User u)
{
    u.Role = 0;
}

static void Main(string[]
args)
{
    User u = new User();
    u.Role = -999;

    SetRoleToZero(u);

    Console.WriteLine(u.Role);
}
```



Подмяна на обект


```
static void
SetRoleToZero(User u)
{
    User newu = new User();
    newu.Role = 0;

    u = newu;
}

static void Main(string[]
args)
{
    User u = new User();
    u.Role = -999;

    SetRoleToZero(u);

    Console.WriteLine(u.Role);
}
```



Редакция на адрес


```
static void
SetRoleToZero(ref User u)
{
    User newu = new User();
    newu.Role = 0;

    u = newu;
}

static void Main(string[]
args)
{
    User u = new User();
    u.Role = -999;

    SetRoleToZero(ref u);

    Console.WriteLine(u.Role);
}
```



C# иска да гарантира, че знаем, че искаме поведение, различно от това по подразбиране. За това иска модификатора да се изписва и при декларация и при извикване.

Задача: Нека функцията `ValidateUserInput()` да попълва цял обект `User`, подаден като параметър, с данните на намерения* потребител:

- Добавете параметър на функцията `ValidateUserInput` от тип `User`.
- Добавете задаване на стойност на новия параметър - нека връща данните на `UserData.TestUser`
 - Или подавате инстанциран обект и копирате всички полета едно по едно
 - Или подавате празен обект и му присвоявате `TestUser` като за параметъра използвате `ref` или `out`.

** Като добавим търсене на потребител, ще попълваме намерения. За сега - `TestUser`.
Функцията продължава да връща `bool`, защото така ще е по-лесно където я викаме да разберем дали валидацията е успешна или не.*

1.1.4. Локални променливи

В C# локалните променливи не могат да бъдат използвани, преди да са инициализирани.

- Инициализацията може и да е `null`.

Тъй като не можем само да обявим указател към типа и да го използваме за инициализация по-късно или в друг метод. Трябва да разполагаме с цял обект. Или `null`.

Задача: Променете метода `Main` :

- Извикването на конструктора `LoginValidation()`:
 - Попитайте потребителя за име и парола и прочетете въведеното (`Console.ReadLine()`)
 - Подайте прочетеното като параметри на новия конструктор `LoginValidation(,)`.
- Извикването на функцията `ValidateUserInput()`:
 - Създайте обект от тип `User` който да подадете на функцията като параметър (Не подавайте попълнен обект. Идеята е `ValidateUserInput()` да го попълни, или да го остави `null` ако валидацията е неуспешна)
 - Ако копирате всички полета едно по едно във `ValidateUserInput()`, то подайте инстанциран обект
 - Ако използвате `ref` или `out` параметър, подайте `null` обект.
- извеждането на данните на потребителя
 - **да не е** от обекта `UserData.TestUser`, а да е от обекта, който подадохме на `ValidateUserInput()`

В момента кодът винаги ще се изпълнява с произволни потребителско име и парола, защото `ValidateUserInput()` винаги връща `true`. По-нататък ще добавим смислени проверки.

- Стартирайте програмата с F5 или зеления триъгълник.

1.2. Типове в C#

В C# типовете са структури или класове.

Тип	Дефиниция
<code>char</code>	<code>public struct Char : ...</code>

	{ ... }
bool	public struct Boolean : ... { ... }
Int	public struct Int32: ... { ... }
double	public struct Double: ... { ... }

По подразбиране структурите се предават по стойност.

Повече за функционалността на различни типове ще разгледате в следващото упражнение.

1.2.1. Наследяване в C#

Класовете и структурите в C# са различни:

- Класовете могат да наследяват само един базов клас
- Класовете могат да наследяват множество интерфейси
- Структурите не поддържат наследяване
- Структурите нямат потребителски конструктори
- Структурите могат да имат член-променливи

1.2.2. Изброени типове в C#

Всички `enum` в C# са `scoped`.

В C# достъпа до членове на `enum` става с оператор `.` (точка), вместо `::`

Пример:

```
UserRoles ur = UserRoles.ADMIN; // Вярно
UserRoles ur = ADMIN; // Грешно
```

Напомняне: изброените типове вътрешно са реализирани чрез `Integer`.

Задача: Променете функцията `ValidateUserInput()` да променя статичното публично свойство `currentUserRole` (от тип `UserRoles`) според `Role` на използвания потребител:

- `currentUserRole = (UserRoles)user.Role;`
(точно преди `return true;`)
(вашият параметър от тип `User` може да е кръстен различно от `user`)

1.2.3. Низове в C#

В C# `string` е клас.

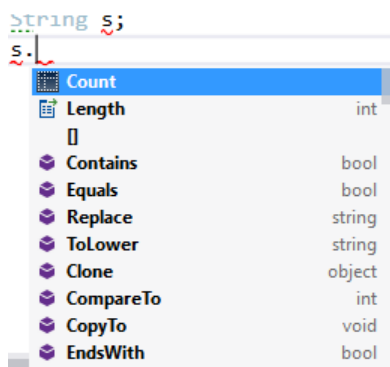
т.е. всяка променлива от тип `string` има вградени член-методи и свойства за манипулация.

```
public sealed class String : ...
{
    ...
}
```

Често използвани методи и свойства:

- За сравнения: [Compare\(\)](#), [CompareOrdinal\(\)](#), [CompareTo\(\)](#), [Equals\(\)](#), [EndsWith\(\)](#), [StartsWith\(\)](#)
- За индексирание: [IndexOf\(\)](#), [IndexOfAny\(\)](#), [LastIndexOf\(\)](#), [LastIndexOfAny\(\)](#)
- За копиране на низ или част от низ: [Copy\(\)](#) , [CopyTo\(\)](#)
- За достъп до част от низ: [Substring\(\)](#), [Split\(\)](#)
- За обединяване на низове: [Concat\(\)](#) , [Join\(\)](#)
- За редакция: [Insert\(\)](#), [Replace\(\)](#), [Remove\(\)](#), [PadLeft\(\)](#), [PadRight\(\)](#), [Trim\(\)](#), [TrimEnd\(\)](#), [TrimStart\(\)](#)
- За посимволна редакция: [ToLower\(\)](#), [ToLowerInvariant\(\)](#), [ToUpper\(\)](#), [ToUpperInvariant\(\)](#)
- За форматиране: [Format\(\)](#)
- За дължина: [Length](#)
- За състояние: [Empty](#), [IsNullOrEmpty\(\)](#)

Тези методи и свойства може да намерите директно в средата:



Освен методите за сравнение на низове се използват и операторите `==` и `!=`. Двамата оператора са чувствителни към малки и главни букви.

Пример: Да добавим проверка в `ValidateUserInput()` дали потребителя е въвел нещо за потребителско име и парола. Ако не е, ще върнем `false`:

- Потребителското име и паролата вече са прехвърлени в частни променливи на класа `LoginValidation`
(вие може да сте ги кръстили различно)
- Остава само да напишем `if` използвайки подходяща функция или свойство на класа `String`
- Трябва да попълним и низът за грешки с подходящо описание:
(вие може да сте ги кръстили различно)

```
Boolean emptyUserName;  
emptyUserName = username.Equals(String.Empty);
```

```

if (emptyUserName == true)
{
    _errorLog = "Не е посочено потребителско име";
    return false;
}

Boolean emptyPassword;
emptyPassword = password.Equals(String.Empty);
if (emptyPassword == true)
{
    _errorLog = "Не е посочена парола";
    return false;
}

```

Empty е статично свойство. За това го викаме директно на класа `String.Empty`, а не на обект (напр. `_username.Empty` е грешка).

Задача: Добавете в `ValidateUserInput` още проверки - дали потребителя е въвел потребителско име и парола по-къси от 5 символа. Ако е - върнете `false`. Попълнете и низът за грешки с подходящо описание.

- Коя функция или свойство ще използвате като най-подходящо?

Идеята е, че ако потребителя не е попълнил адекватно данните, няма смисъл да правим по-сложни проверки.

Проверка дали потребителското име и паролата са верни ще добавим след малко.

- Стартирайте програмата с F5 или зеления триъгълник.

1.2.4. Масиви в C#

В C# `array` е клас.

т.е. всяка инстанция на масив има вградени член-методи и свойства за манипулация.

```

public abstract class Array : ...
{
    ...
}

```

- В C# декларацията на масив има друг синтаксис: Скобите `[]` са част от типа.
- Тъй като масивът е обект, има нужда от инстанция.

Пример:

```

string[] names;
names = new string[2];

names[0] = "Иванчо";
names[1] = "Марийка";

int[] numbers = new int[5];
User[] TestUsers = new User[5];

```

Пример: Въпреки, че сме задали размерност на масива, това не заделя памет за самите елементи.

```
string[] names = new string[2];  
names[0].Equals("Петърчо"); // Грешка   names[0] == null  
  
User[] TestUsers = new User[5];  
TestUsers[0].Username = "admin"; // Грешка   TestUsers[0] == null
```

Защото всички типове, които можем да поместим в масив, са класове или структури. Дори и базовите.

Задача: Нека добавим повече примерни потребители в системата:

- Променете свойството `UserData.TestUser` на масив с име `TestUsers`:
(Нали не ви притеснява, че "цял" масив ще е свойство?)
- Променете и частното поле `_testUser` да е масив с ново име.
- Променете `get` метода.
- Променете метода `ResetTestUserData()` да попълва `TestUsers` с 3 различни потребителя.
(Един администратор и 2 студента)
(Не забравяйте да им залагате имена и пароли не по-къси от 5 символа)

Задача: Добавете функция `IsUserPassCorrect` в класа `UserData`, която връща обект от тип `User`:

- Добавете два параметъра за потр. име и парола.
- Добавете проверка с цикъл `for` дали въведените потр. име и парола са сред наличните в `UserData.TestUsers`
- Дължината на масива може да намерите със свойството `Length`.
(Дължината на масива не отговаря на последния индекс, нали така?)
- Ако се намери такъв потребител да се върне като резултат от функцията.
- Ако не се намери - да се върне `null`.

Задача: Променете метода `ValidateUserInput` в класа `LoginValidation`.

Вече няма да се обръщаме директно към `UserData.TestUser`. Резултатът ще е зависим единствено от намерен потребител по потребителско име и парола.

- Добавете още една проверка с обръщение към функцията `IsUserPassCorrect(,)`.
(Тази проверка трябва да е след предните проверки, за да не се изпълнява напразно)
- Да попълва параметър от тип `User` с върнатото от функцията `IsUserPassCorrect(,)`.
- Да връща `true` само ако всички проверки са успешни.
(т.е. върнатия обект не е `null`)
- Да попълва `currentUserRole` с ролята на намерения потребител само ако има такъв.
(т.е. върнатия обект не е `null`)
- Ако проверката е неуспешна да попълва низът за грешки с подходящо описание.
- Ако някоя от проверките е неуспешна да попълва `currentUserRole` с ролята `ANONYMOUS`.

➤ Стартирайте програмата с F5 или зеления триъгълник.

1.3. Нови запазени думи в C#

1.3.1. Блок finally

В C# може да добавите finally блок след try блока, в който да се помести код, който е нужно да се изпълни дори когато е възникнало изключение.

Пример:

<pre>try { result = num1 / num2; } catch (DivideByZeroException e) { Console.WriteLine ("Exception caught"); }</pre>	<pre>try { result = num1 / num2; } catch (DivideByZeroException e) { Console.WriteLine ("Exception caught"); } finally { result = 0; }</pre>	<pre>try { result = num1 / num2; } finally { result = 0; }</pre>
---	---	--

Повече управление на изключения ще разгледате в следващото упражнение при работа с файлове.

1.3.2. Условен оператор switch

В C# оператора `switch` не позволява изпълнението от една `case` секция автоматично да продължи със следващата.

т.е. сме длъжни да напишем `break`;

Пример:

```
int caseSwitch = 1;

switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

Задача: Променете Main метода:

- След като изкара данните за логния потребител (ако е успешно логнат):
 - Използвайте `switch` оператора.
 - Променете изведеждането на екрана на стойността на `LoginValidation.currentUserRole`
 - Изведете подходящ текст в зависимост от стойността на ролята (т.е. за всяка роля да излиза различен четим текст)

- Стартирайте програмата с F5 или зеления триъгълник.

2. Цикъл foreach

В C# цикълът `foreach` е по-лесна за използване и по-четима алтернатива на цикъла `for`.

```
foreach (Type element in iterable_item)
{
    // тяло на цикъла, в което
    // работим директно с element
}
```

- Итераторът `element` може да го кръстим както решим. Той ще представлява текущия елемент от масива `iterable_item` в тялото на цикъла.
- Типът `Type` от който декларираме итераторът `element` трябва да е тип, съвместим с типа на елементите в масива `iterable_item`:

```
Type[] iterable_item;
```

Пример: Следните два цикъла са еквивалентни:

<pre>char[] myArray = { 'H','e','l','l','o' }; for (int i=0; i < myArray.Length; i++) { Console.WriteLine(myArray[i]); }</pre>	<pre>char[] myArray = { 'H','e','l','l','o' }; foreach (char ch in myArray) { Console.WriteLine(ch); }</pre>
---	--

Грубо казано `myArray[i] == ch` вътре в тялото.

Спестява изписването на името на масива всеки път и ръчната работа с индексите.

Задача: Променете функцията `UserData.IsUserPassCorrect` да използва цикъл `foreach`.

- Стартирайте програмата с F5 или зеления триъгълник.

3. Delegates

Делегатите в C# са типове и служат за същата цел като указатели към функция в C++.

Защото няма указатели.

Т.е. казваме коя функция ще бъде представявана от делегата и после викаме делегата, а ще се изпълни присвоената функция.

Делегатите са повече от указател към функция.

Те са строго типизирани и безопасни, защото:

- С дефиницията на делегат се ограничават функциите, които може да съдържа.
- Само метод със същата сигнатура като делегата може да му се присвои.

След след като е дефиниран, за да бъде използван, делегатът трябва да се инстанцира с **new**.

Пример:

Функции със същата сигнатура

```
delegate int NumberChanger(int n, int m);

public static int AddNum(int p, int q)
{
    int num = p + q;
    return num;
}
public static int MultNum(int r, int s)
{
    int num = r * s;
    return num;
}

static void Main(string[] args)
{
    NumberChanger nc1 = new
        NumberChanger(AddNum);
    NumberChanger nc2 = new
        NumberChanger(MultNum);

    int i;
    //calling the methods using the
        delegate objects

    i = nc1(5, 5);
    Console.WriteLine("Value of i: " + i);
    i = nc2(5, 5);
    Console.WriteLine("Value of i: " + i);
}
Value of i: 10
Value of i: 25
```

Грешка при функции с различна сигнатура

```
delegate int NumberChanger(int n, int m);

public static int Square(int j)
{ return j * j; }

static void Main(string[] args)
{
    NumberChanger nc3 = new
        NumberChanger(Square); // Грешка

    }
}

CS0123 No overload for 'Square' matches delegate 'Program.NumberChanger'
```

Задача: Да позволим на ползвателя на класа `LoginValidation` да определя как да се съобщи за неуспешна валидация:

Редактирайте класа `LoginValidation` по следния начин:

- Добавете делегат:

```
public delegate void ActionOnError(string errorMsg);
```

(Дефиницията на делегат е дефиниране на нов тип. В случая е вътре в класа)

- Добавете частно поле от типа на делегата `ActionOnError`.
- Добавете параметър от типа на делегата `ActionOnError` към конструктора.
- Инициализирайте новото частно поле с новия параметър в конструктора.
- Извиквайте новото частно поле от тип `ActionOnError` във функцията `ValidateUserInput()` тогава, когато се установи грешка във валидацията
 - То е поле (член-променлива), но съхранява функция. За това я викайте като функция.
 - Може би най-удачно е да подавате като параметър полето с описанието на

грешката.

(Ако сте кръстили новото поле например `_errorfunc`, то го викайте така `_errorfunc(_errorLog);`)

По-нататък ще използваме този клас от визуално приложение. Там няма да пишем в конзолата, а ще подадем друга функция.

Задача: Да подадем на новия конструктор на `LoginValidation` функция, която да се изпълнява при грешка.

Редактирайте класа `Program` по следния начин:

- Създайте нова функция, която да отговаря на сигнатурата на типа `ActionOnError`:



```
public delegate void ActionOnError(string errorMsg);
```

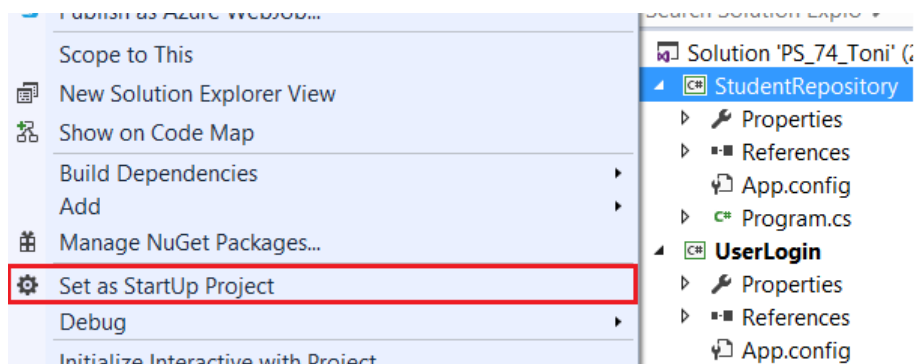
- т.е. дад връща `void`
- и да приема един параметър от тип `string`
- Напишете подходящо тяло на функцията (например `Console.WriteLine("!!! " + s + " !!!");`)
- В `Main` метода подайте името на новата функция на конструктора на `LoginValidation`.

- Стартирайте програмата с F5 или зеления триъгълник.

Общи задачи (пример за работа в час):

Задача: Добавяме нов проект към решението (solution-a):

- ("File" → "Add" → "New Project ...")
(Или от "Solution Explorer" → "Solution" →  → "Add" → "New Project ...")
- В диалоговия прозорец изберете "Visual C#" → "Windows" → "Console Application"
- Кръстете новия проект **StudentRepository**.
- В Solution Explorer с десен бутон на новия проект изберете "Set as StartUp Project"
("Solution Explorer" → "StudentRepository" →  → "Set as StartUp Project")



- Името на проекта за стане удебелено.

При наличие на повече от един проект Visual Studio трябва да знае кой да стартира при натискане на бутона Start (F5). Ще стартира StartUp проекта.

Добра идея е да затворите всички файлове отворени за редакция от другия проект, за да не става объркване.

В проекта **StudentRepository** (оттук надолу):

Задача: Създайте клас с име **Student**

- Разширете класа **Student** с публични полета от подходящ тип и име, които да съхраняват:
 - Име
 - Презиме
 - Фамилия
 - Факултет
 - Специалност
 - Образователно-квалификационна степен
 - Статус (заверил, прекъснал, семестриално завършил,...)
 - Факултетен номер
 - Курс
 - Поток
 - Група

Ако създавате класа през Solution Explorer внимавайте на кой проект давате десен бутон.

Ако създавате класа от "Project" → "Add New Item..." внимавайте кой проект е селектиран в Solution Explorer.

Какви типове са подходящи за полетата?

Каква е подходяща видимост на полетата?
Има ли нужда от специална реализация (конструктори, методи)?

Задача: Създайте клас с име `StudentData`.

В него:

- Добавете свойство `TestStudent` от тип `Student`
- Свойството трябва да може да се чете от вън и да се променя само в класа.

Подходящо ли е нещо да е статично?

Каква е подходящата видимост?

Запишете си проекта! USB, DropBox, GoogleDrive, e-mail, SmartPhone, където и да е.

До края на упражненията ще работите върху този проект.

Вкъщи също ще работите върху този проект.

Накрая трябва да го представите завършен.