

Упражнение №2 по ПС - Част 2

.Net

System
System.Collections
System.IO
System.Text
System.Linq

Целта на това упражнение:

Запознаване с най-често използваните класове на рамката .Net.

Необходими познания:

От предметите дотук и предното упражнение трябва да можете:

- Да имате основни умения с езика C#
- Да имате идея за интерфейси.
- Да имате понятие за файлова система.
- Да имате понятие за база данни и SQL.

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.

Задачите в упражнението изграждат:

Университетска информационна система.

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

- разширяваме класа User с дата на създаване и дата на валидност на потребителя

- добавяме меню за администратор, което е достъпно след успешна регистрация

```
Изберете опция:  
0: Изход  
1: Промяна на роля на потребител  
2: Промяна на активност на потребител  
3: Списък на потребителите  
4: Преглед на лог на активност  
5: Преглед на текуща активност
```

- добавяме функционалност, която да реализира опциите от менюто

- нов клас, който да пази лог на действията на потребителя (Logger). Логът също така се запазва в текстов файл.

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.

Ако има администраторски привилегии ще му даде достъп до меню с опции за редакция на потребители и проследяване на активността на приложението.

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

*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. (Така че направете **UserLogin** да е StartUp проект.)*

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

1. Основни класове в System.

В именното пространство System се намират основни класове и базови класове, с които се дефинират най-често използваните типове, събития, интерфейси и изключения.

В System са дефинирани:

char
bool
Int
double

Надолу следват техни функционалности и други типове от System.

1.1. DateTime

Типът **DateTime** представлява момент във времето. Най-често представен като дата и час.

- Типът **DateTime** е структура:

```
public struct DateTime : IComparable, IFormattable, IConvertible, ISerializable,
    IComparable<DateTime>, IEquatable<DateTime>
```

По подразбиране структурите се предават по стойност.

1.1.1. Конструктори

Типът **DateTime** има дефинирани множество конструктори. Разгледайте ги директо в средата:

```
DateTime dt = new DateTime(| );
```

▲ 4 of 12 ▼ **DateTime**(int year, int month, int day)

Initializes a new instance of the **DateTime** structure to the specified year, month, and day.
year: The year (1 through 9999).

↑ Ctrl + Shift + Space

Пример: Следния конструктор задава само датата 3 март 1878 г. на обект от тип **DateTime**:

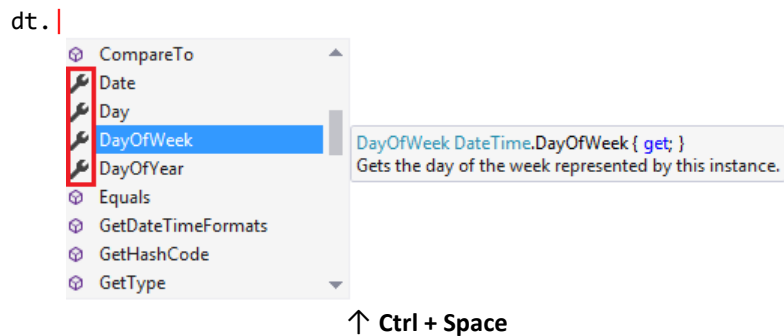
```
DateTime dt = new DateTime(1878, 3, 3);
```

Задача: Намерете и използвайте конструктора, който ще ви позволи да зададете следната информация:

15 септември 2017 г., 10 ч. 30 мин. (0 сек.)

1.1.2. Свойства

Обектите от тип `DateTime` имат множество свойства с които можем да достъпваме информацията в тях в различни форми. Разгледайте ги директо в средата:



Пример: Свойството `DayOfWeek` представя какъв ден е датата в обекта:

```
DateTime dt = new DateTime(1878, 3, 3);  
DayOfWeek day = dt.DayOfWeek;  
Console.WriteLine(day);
```

Sunday

Различните свойства са от различни типове. В случая има специализиран тип структура `DayOfWeek`.

Задача: Намерете и използвайте свойството, което ще ви предостави само часа, записан в обекта:

15 септември 2017 г., 10 ч. 30 мин. (0 сек.)

1.1.3. Статични свойства

Класът `DateTime` предоставя статични свойства с полезни стойности при работа с време. Следните две са най-значими и връщат обекти попълнени с текущия момент или днешна дата:

```
DateTime.Now;  
DateTime.Today;
```

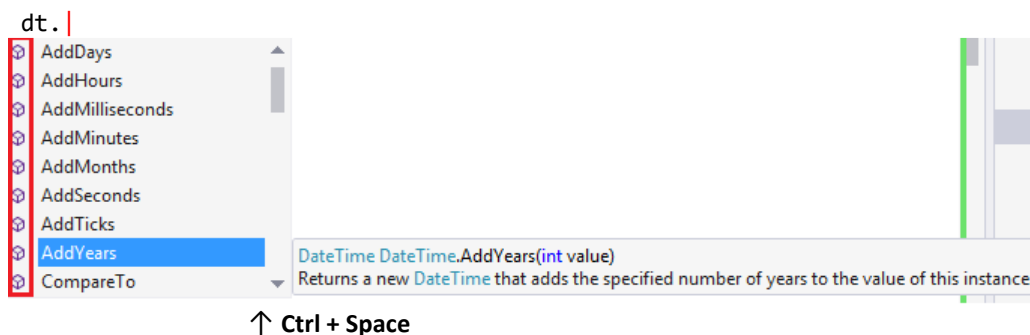
Пример: Да намерим какъв ден е днес:

```
DateTime dt = DateTime.Today;  
DayOfWeek day = dt.DayOfWeek;  
Console.WriteLine(day);
```

Задача: Намерете кой час тече в момента.

1.1.4. Методи

Предоставяните функции може да видите директно в средата:



Пример: Да намерим коя е следващата и предишната година:

```
DateTime dt = DateTime.Today;
DateTime dtnextyear = dt.AddYears(1);
int nextyear = dtnextyear.Year;

DateTime dtlastyear = dt.AddYears(-1);
int lastyear = dtlastyear.Year;
```

AddYears(1) е функция, която връща нов обект от тип `DateTime`, а не променя обекта към когото е извикана.

Пример: Да формираме обект `DateTime` обект представляващ 1 януари следващата година:

```
DateTime dt = DateTime.Today;
int nextyear = dt.AddYears(1).Year;

DateTime firstnextyear = new DateTime(nextyear, 1, 1);
```

Задача: Намерете коя дата ще е след 12 часа.

Задача: Добавете дата на създаване на потребител.

- Добавете публично поле `Created` от тип `DateTime` към класа `User`.
- Променете метода `UserData`. `ResetTestUserData()` да задава текущо време за всеки потребител в новото поле.

1.2. MinValue и MaxValue

Всички числови типове предлагат публични полета на `MaxValue` и `MinValue`. Тези полета съдържат най-голямата и най-малката стойност, която може да се съдържа в рамките на типа без опасност от препълване.

Числовите типове, като структури, не могат да са null и винаги имат някаква стойност. Т.е. полета на MaxValue и MinValue са удобни за инициализация.

MaxValue и MinValue са статични публични полета на самия тип.

Пример:

```
Int32 i = Int32.MaxValue;
Int32 i = Int32.MinValue;
Double d = Double.MaxValue;
Double d = Double.MinValue;
DateTime t = DateTime.MaxValue;
DateTime t = DateTime.MinValue;
```

Пример: Да намерим най-голямата от три дати:

```
DateTime[] datearray = new DateTime[3];
datearray[0] = new DateTime(1998, 1, 1);
datearray[0] = new DateTime(2017, 6, 1);
datearray[0] = new DateTime(2017, 1, 1);

DateTime maxdate = DateTime.MinValue;
for (int i = 0; i < datearray.Length; i++)
{
    if (maxdate < datearray[i])
        maxdate = datearray[i];
}
```

Задача: Добавете дата до кога е активен потребител.

- Променете класа.
- Променете метода `UserData.ResetTestUserData()`. Нека всички тестови потребители да са валидни завинаги.

1.3. Type Conversions

- Тъй като C# е строго типизиран при компилация
- След декларация на променлива тя не може да бъде предекларирана с друг тип
- Декларирана променлива не може да съдържа друг тип
- За присвояването на друг тип е необходимо да е заложено конвертиране

Например няма нито конвертиране нито *cast* между целочислен тип и низ.

1.3.1. Класът Convert

Класът `System.Convert` е помощен за прехвърляне между несъвместими типове.

```
string str = "true";
bool flag = Convert.ToBoolean(str);
```

Задача: Изчетете написана дата от конзолата и я запишете в обект от тип `DateTime`.

1.3.2. ToString() и Parse()

В частност за конвертиране от и към низ съществуват и функциите `ToString()` и `Parse()`.

- `ToString()` е функция, описана още в класът `Object`, от който наследяват всички класове.
- Стандартната имплементация връща наименованието на класа
- `ToString()` се вика по подразбиране, когато обект е подаден като низ
- За базовите типове `ToString()` е предефинирана да изписва стойността

Пример: Следните два блока са еквивалентни:

```
User u = new User();
String s1 = u.ToString();
Console.WriteLine(u);
```

```
Int32 i = 5;
String s2 = i.ToString();
Console.WriteLine(i);
```

```
DayOfWeek day = DayOfWeek.Friday;
String s3 = day.ToString();
Console.WriteLine(day);
```

```
User u = new User();
String s1 = u.ToString();
Console.WriteLine(u.ToString());
```

```
Int32 i = 5;
String s2 = i.ToString();
Console.WriteLine(i.ToString());
```

```
DayOfWeek day = DayOfWeek.Friday;
String s3 = day.ToString();
Console.WriteLine(day.ToString());
```

```
UserLogin.User
5
Friday
```

- `Parse()` е функция налична в числовите типове
- `Parse()` изчита съдържанието на низ и конвертира стойността към типа, за когото се вика
- `Parse()` е наличен и за изброени типове

Пример: Следните два блока са еквивалентни:

```
String s1 = "01.01.2019 00:00:00";
DateTime dt = DateTime.Parse(s1);
```

```
String s2 = "5";
Int32 i = Int32.Parse(s2);
```

```
String s1 = "01.01.2019 00:00:00";
DateTime dt = Convert.ToDateTime(s1);
```

```
String s2 = "5";
Int32 i = Convert.ToInt32(s2);
```

Задача: Да добавим функционалност, която да позволява на администратор да променя ролята и активността на потребител.

- Добавете метод `SetUserActiveTo` в класа `UserData`, който да приема два параметъра - от тип `string` за потребителско име и `DateTime` за нова дата на активност.
 - Методът трябва да намери потребителя с подаденото име и да му промени датата, до която е активен, с подадената дата.
Какъв трябва да е метода? Статичен или не? Публичен или не?
- Добавете метод `AssignUserRole` в класа `UserData`, който да приема два параметъра от тип `string` за потребителско име и `UserRoles` за новата роля.

- Методът трябва да намери потребителя с подаденото име и да му промени ролята с подадената.
Какъв трябва да е метода? Статичен или не? Публичен или не?
- Добавете функционалност в класа `Program`, която да се извиква в `Main` метода:
 - След успешно логване да запитва коя функционалност да се извика.
(Меню за администратор)

```
Изберете опция:
0: Изход
1: Промяна на роля на потребител
2: Промяна на активност на потребител
```

- При извикване на `SetUserActiveTo` трябва да се въведе дата.
- При извикване на `AssignUserRole` трябва да се въведе нова роля
Може да се въведе `int`.
- И в двата случая трябва да се посочи потребителско име на потребителя, който ще се променя.
- Функционалността за администратор след регистрация е може би подходящо да е в отделен метод в класа `Program`.

2. Основни класове в `System.Collections`

В именното пространство `System.Collections` се намират типове за стандартни, специализирани и общи колекции от обекти.

2.1. Класът `List`

Класът `List` е реализацията на списък в `.Net`

- Списъкът е строго типизиран.
- Типът на обектите в списъка се подава с шаблон `List<T>`

```
List<int> integers;
List<string> colors;
```

- Тъй като целият списък представлява обект на клас е необходима инициализация

```
List<int> integers = new List<int>();
List<string> colors = new List<string>();
```

- Добавянето на елементи в края на списъка става с `Add`:

```
List<int> integers = new List<int>();
integers.Add(2);
integers.Add(3);
integers.Add(5);
integers.Add(7);
```

```
List<string> colors = new List<string>();
colors.Add("Red");
colors.Add("Blue");
colors.Add("Green");
```

Когато добавяме с `Add()` размерът на списъка се увеличава с един елемент и се добавя новия елемент.

- Предоставя достъп по индекс.
- Размерът на списъка можем да вземем с Count

<pre> integers[0] = 2; integers[1] = 3; integers[2] = 5; integers[3] = 7; </pre>	<pre> for (int i = 0; i < colors.Count; i++) { Console.WriteLine(colors[i]); } </pre>	<pre> foreach (string color in colors) { Console.WriteLine(color); } </pre>
----------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------

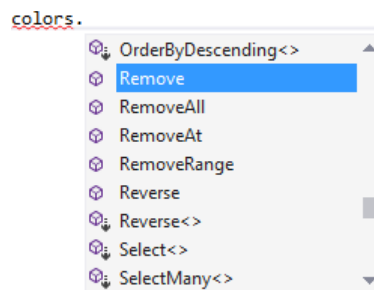
Свойството `Capacity`, което може да се зададе и с конструктора, не създава празни елементи в списъка, т.е. списъка не се държи като масив. За да достъпим елемент с индекс 4, например, първо трябва да сме добавили 5 елемента в списъка, дори и да сме задали по-голямо `Capacity`.

```

List<int> integers = new List<int>(5);
integers[4] = 60; // Грешка

```

- Друга функционалност на списъка може да видите директно в средата:



Някои полезни функции са:

```

.Insert();
.Sort();
.Remove();
.Contains()
.Clear();

```

Пример: Да добавим функционалност за логване действията на потребител.

- Трябва да създадем нов статичен клас

```

static class Logger
{
}

```

- Към класа добавяме ново частно поле-списък `List<string>`, в който ще записваме всички действия на текущия потребител. Трябва и да го инстанциираме.

```

static private List<string> currentSessionActivities = new List<string>();

```

- Към класа добавяме и метод `LogActivity`, който ще записва действията.

```

static public void LogActivity(string activity)

```

- В метода `LogActivity` добавяме по един ред в списъка с действия `currentSessionActivities`:

```

string activityLine = DateTime.Now + ";";

```

```

+ LoginValidation.currentUserUsername + ";"
+ LoginValidation.currentUserRole + ";"
+ activity;
currentSessionActivities.Add(activityLine);

```

- Сега само трябва да добавим извикване на метода в методите ValidateUserInput, AssignUserRole, и SetUserActiveTo:

```

Logger.LogActivity("Успешен Login"); //във ValidateUserInput
Logger.LogActivity("Промяна на роля на " + username); //в AssignUserRole
Logger.LogActivity("Промяна на активност на " + username); //в SetUserActiveTo

```

Задача: Променете масивът `UserData.TestUsers` да бъде списък.

- Променете функциите `ResetTestData`, `IsUserPassCorrect`, `AssignUserRole`, `SetUserActiveTo` да работят със списък.

2.2. Класът Dictionary

Класът `Dictionary` в C# реализира списък от двойки ключ-стойност.

- Ключът е ключова стойност, по която се открива стойността.
- Обект от всякакъв тип може да служи за ключ.
- Не може да има дублиращи се или null ключове.
- Стойностите може да се дублират.

Т.е. речникът е списък, при който индексът може да бъде нецелочислен.

- Списъкът е строго типизиран.
- Типът на обектите в речника се подава с шаблон `Dictionary<TKey, TValue>`

```
Dictionary<string, int> dict;
```

- Тъй като целият речник представлява обект на клас е необходима инициализация

```
Dictionary<string, int> dict = new Dictionary<string, int>();
```

- Добавянето на елементи към речника става с `Add`:

```

Dictionary<string, int> dict = new Dictionary<string, int>();
dict.Add("first", 1);
dict.Add("1st", 1);
dict.Add("I", 1);
dict.Add("second", 2);
dict.Add("2nd", 2);
dict.Add("II", 2);

```

- Достъпа до стойностите става по ключ:

```

int j = dict["first"];
int k = dict["2nd"];

```

- Възможен е и достъп по индекс с функцията `ElementAt()`.
- Съдържаните елементи са от тип `KeyValuePair<TKey, TValue>`
- Размерът на речника можем да вземем с `Count`

```
for (int j = 0; (j < dict.Count); j++)
{
    Console.WriteLine(dict.ElementAt(j));
}

foreach (KeyValuePair<string, int> item in dict)
{
    Console.WriteLine(item);
}
```

```
[first, 1]
[1st, 1]
[I, 1]
[second, 2]
[2nd, 2]
[II, 2]
```

- Достъпа до съставните ключ и стойност става през свойствата `Key` и `Value`:

```
for (int j = 0; (j < dict.Count); j++)
{
    Console.WriteLine(dict.ElementAt(j).Key);
    Console.WriteLine(dict.ElementAt(j).Value);
}

foreach (KeyValuePair<string, int> item in dict)
{
    Console.WriteLine(item.Key);
    Console.WriteLine(item.Value);
}
```

```
first
1
1st
1
I
1
second
2
2nd
2
II
2
```

- Друга функционалност на списъка може да видите директно в средата.

Пример: Да добавим функция, която да изписва имената на всички потребители в системата.

- Подходящо е функцията да връща речник, за да може да достъпваме потребителите директно чрез потребителско име.
- Речникът е съставен от `string` за потребителско име като ключ и `int` за индекса на потребителя в `TestUsers` като стойност.
- Добавяме функцията с име `AllUsersUsernames` към класа `UserData`:

```
static public Dictionary<string, int> AllUsersUsernames()
{
}
```

- Функцията трябва да връща резултат. Добавяме си обект в нея, който ще върнем накрая:

```
Dictionary<string, int> result = new Dictionary<string, int>();
...
return result;
```

- Накрая попълваме речника от списъка `TestUsers`:

(вашият полета може да са кръстени различно)

```
...  
for (int i = 0; i < TestUsers.Count; i++)  
{  
    result.Add(TestUsers[i].Username, i);  
}  
...
```

Задача: Да използваме функцията `AllUsersUsernames()`, която току-що написваме:

- Добавете в менюто за администратора (функционалността в класа `Program`, която да се извиква в `Main` метода след успешно логване) още една опция да изпише всички потребители

```
Изберете опция:  
0: Изход  
1: Промяна на роля на потребител  
2: Промяна на активност на потребител  
3: Списък на потребителите
```

- Изполвайте резултата от функцията `AllUsersUsernames()`:

```
Dictionary<string, int> allusers = UserData.AllUsersUsernames();
```

- За всеки обект в речника изведете неговия ключ:

```
Console.WriteLine(user.Key);
```

- Добавете в менюто за администратора (функционалността в класа `Program`, която да се извиква в `Main` метода след успешно логване) при избор на опциите извикващи `SetUserActiveTo` или `AssignUserRole` :

- За да намерите индекса на потребителското име посочено за редакция използвайте резултата от функцията `AllUsersUsernames()`:

```
Dictionary<string, int> allusers = UserData.AllUsersUsernames();  
string usertoedit = Console.ReadLine();
```

```
...
```

```
UserData.AssignUserRole(allusers[usertoedit], NewUserRole);  
UserData.SetUserActiveTo(allusers[usertoedit], NewDateTime);
```

- Променете функциите `AssignUserRole` и `SetUserActiveTo` да приемат индекс на потребителя вместо името на потребителя.

3. Основни класове в System.IO

В именното пространство `System.IO` се намират типове, позволяващи четене и запис във файл и в поток, а също и типове за работа с файлове и директоии.

`System.IO` не се включва по подразбиране. Трябва да добавите:

```
using System.IO;
```

3.1. Класът File

Класът File предлага **статични** методи за манипулация на единичен файл.

- С функцията Exists проверяваме дали има такъв файл:

```
File.Exists("test.txt");
```

- С функцията ReadAllText достъпваме съдържанието във файла.

```
File.ReadAllText("test.txt");
```

- С функцията WriteAllText задаваме ново съдържание на файла.

```
File.WriteAllText("test.txt", s);
```

- С функцията добавяме съдържание към края на файла.

```
File.AppendAllText("test.txt", s);
```

Пример:


Четене и запис на файл:

```
if (File.Exists("test.txt") == true)
{
    string s = File.ReadAllText("test.txt");
    Console.WriteLine(s);
    File.WriteAllText("test.txt", s);
}
```


Добавяне към файл:

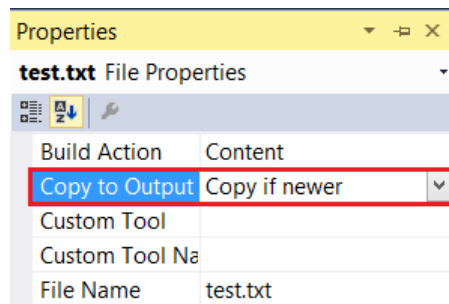
```
if (File.Exists("test.txt") == true)
{
    string s = Console.ReadLine();
    File.AppendAllText("test.txt", s);
}
```

Пример: Да добавим файл, който можем да използваме за горните примери:

- Вариант 1: Задаваме пълен път до файла, напр. **"D:/test.txt"**
(Това може да създаде проблеми с преносимостта)
- Вариант 2: Добавяме файл към проекта:
 - ("Project" → "Add New Item...")
(Или от "Solution Explorer" → "Project" →  → "Add" → "New Item...")
 - В диалоговия прозорец избягете "Visual C# Items" → "General" → "Text File"



- Кръстете новия файл test.txt
- В Solution Explorer с десен бутон на новия файл поискйте Properties
("Solution Explorer" → "test.txt" →  → "Properties")
- От прозореца Properties, в който ще ви прехвърли, настройте свойството "Copy to Output" на "Copy if newer"



(Указвате на Visual Studio да премести файла в дебъг директорията, но само ако дебъг версията вече не е попълнила нещо в него)

- Сега може да се обръщате с късо име към файла: **"test.txt"**

Задача: Добавете в метода `Logger.LogActivity()` код, който да добавя редове в лог файл.

3.2. Класовете `StreamReader` и `StreamWriter`

`StreamReader` е помощен клас за изчитане на символи от поток.

`StreamWriter` е помощен клас за запис на символи от поток

- За разлика от `File` не се налага да отваряме и затваряме файла при всяка операция.
- Можем да го отворим веднъж и да четем или пишем продължително.
- Недостатък е, че трябва да затворим файла в края, когато вече не е нужен.

`StreamReader` и `StreamWriter` не са статични класове. Т.е. трябва да инстанцираме обект.

Пример:

```
StreamReader sr = new
    StreamReader("TestFile.txt");

string line = sr.ReadLine();
Console.WriteLine(line);

line = sr.ReadLine();
Console.WriteLine(line);

line = sr.ReadLine();
Console.WriteLine(line);
sr.Close();
```

```
StreamWriter outputFile = new
    StreamWriter("test.txt");

string ss = Console.ReadLine();
outputFile.WriteLine(ss);

ss = Console.ReadLine();
outputFile.WriteLine(ss);

ss = Console.ReadLine();
outputFile.WriteLine(ss);
outputFile.Close();
```

Задача: Добавете функционалност за визуализация на лог файла.

```
Изберете опция:
0: Изход
1: Промяна на роля на потребител
2: Промяна на активност на потребител
3: Списък на потребителите
4: Преглед на лог на активност
```

4. Основни класове в System.Text

Именното пространство System.Text съдържа класове за работа с ASCII и Unicode кодирани символни низове.

4.1. Класът String

Класът String вече го разглеждахме в предното упражнение. Но въпреки, че е клас, се държи различно:

Обектите от тип String са неизменими (immutable).

- Веднъж с назначена стойност, обект от тип String не се променя директно
- Всяка промяна създава нов обект, където се запазва резултата.

Ето защо операциите с низове връщат като резултат обект с резултата. Това се случва индиректни и при операции с предефинираният оператор за конкатенация ($s = s1 + s2$)

4.2. Класът StringBuilder

Класът `StringBuilder` служи за изграждане и промяна на (дълги) низове.

- `StringBuilder` преодолява трудностите свързани с неизменимото (immutable) поведение на класът String
- Промените се изпълняват в рамките на една и съща област от паметта (буфер).

Когато приключим с редакциите, можем отново да работим със String чрез метода `ToString()`.

Пример:

```
StringBuilder sb = new StringBuilder();
sb.Append("Numbers: ");

for (int index = 1; index <= 200000; index++)
{
    sb.Append(index);
}

Console.WriteLine(sb.ToString());
```

Задача: Добавете функционалност за визуализация списъка с текуща активност `Logger.currentSessionActivities`:

```
Изберете опция:
0: Изход
1: Промяна на роля на потребител
2: Промяна на активност на потребител
3: Списък на потребителите
4: Преглед на лог на активност
5: Преглед на текуща активност
```

- Подходящо е да използвате `StringBuilder`.
- Добавете нова функция `GetCurrentSessionActivities()` в класа `Logger`, която да извлича `string` от масива `currentSessionActivities`.
(От менюто за администратора извиквайте новата функция.)

Задача: Променете функционалността за визуализация на лог файла, ако сте я реализирали със цикъл `for` и `string`, да използва `StringBuilder`.

5. LINQ

LINQ = Language Integrated Query

C# е процедурен език, което го прави неудобен при работа с множества (защото трябва да се изпише код за работа с всеки елемент на множеството).

LINQ е декларативно разширение на C# за удобна работа с множества (чрез заявки).

Предимства на LINQ:

- Вграден в езика, подлежи на проверка при компилация
- Единен синтаксис за работа с множества без значение източникът на данни (масив или списък с обекти, мемори таблица, SQL DB, XML файл, др)
- Къс запис и четим код
- Лесен метод за филтрация
- Функционалност за групиране, сортиране, обединяване, пр.

Към момента ще правим заявки само върху масиви и списъци в паметта.

5.1. Query синтаксис с LINQ

- Избираме източникът на данни (списъка) и променливата итератор със запазените думи `from` и `in`
(Синтаксисът е подобен на `foreach`)

```
from line in File.ReadLines("test.txt")
```

- Избираме резултата със запазената дума `select`

```
select line
```

```
from line in File.ReadLines("test.txt") select line
```


- Резултатът от LINQ заявка е от тип `IEnumerable`.
Т.е. може да се преобразува към тип, който имплементира `IEnumerable`, например `List`.

```
IEnumerable<string> lines = from line in File.ReadLines("test.txt") select line;
List<string> lines = (from line in File.ReadLines("test.txt") select line).ToList();
```

Пример: Нека видим как една и съща LINQ заявка връща различен резултат, защото сме указали различни свойства на обекта.

Целия обект:	Свойство Length	Произведен обект, получен с Replace
--------------	-----------------	-------------------------------------

```
StringBuilder sb = new StringBuilder();

List<string> tools = new List<string>()
{ "Tablesaw", "Bandsaw", "Planer", "Jointer", "Drill", "Sander" };

List<string> list = (from t in tools select t).ToList();
foreach (string ss in list)
{
    sb.Append(ss + Environment.NewLine);
}

List<int> list = (from t in tools select t.Length).ToList();
foreach (int ss in list)
{
    sb.Append(ss + Environment.NewLine);
}

List<string> list = (from t in tools select t.Replace("a", "@")).ToList();
foreach (string ss in list)
{
    sb.Append(ss + Environment.NewLine);
}

Console.WriteLine(sb.ToString());
```

```
Tablesaw
Bandsaw
Planer
Jointer
Drill
Sander
```

```
8
7
6
7
5
6
```

```
T@bles@w
B@nds@w
Pl@ner
J@inter
Drill
S@nder
```

- Филтрираме данните със запазената дума `where`

```
where line.Length > 0

from line in File.ReadLines("test.txt") where line.Length > 0 select line

List<string> lines = (from line in File.ReadLines("test.txt")
where line.Length > 0 select line).ToList();
```

Пример: Еднакви заявки с различни условия (по предния пример):

<pre>List<string> list = (from t in tools where t.Contains("a") select t).ToList();</pre>	<pre>List<string> list = (from t in tools where t.Length > 6 select t).ToList();</pre>
---------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------

Пример: Да променим функцията `Logger.GetCurrentSessionActivities()` да работи с LINQ заявка:

- Да добавим параметър на функцията `string filter`
 - Ще ни позволи да визуализираме само някои от действията, указани с филтъра
- Към момента сугурно имате един цикъл `for` или `foreach`, който обхожда масива с действия:
(Ако имате друго - хвала! Но пак за филтъра е най-подходящо да ползвате LINQ заявка, така че разгледайте надолу)

```
StringBuilder sb = new StringBuilder();

foreach (var action in currentSessionActivities)
{
    sb.Append(action);
}
...
```

- Идеята е, вместо да работим с масива `currentSessionActivities`, да работим с подмножество съдържащо само редове, съдържащи филтъра:

```
List<string> filteredActivities = ... (LINQ заявка върху currentSessionActivities)...;

foreach (var action in filteredActivities)
{
    sb.Append(action);
}
```

(Добавете новата променлива-списък и променете цикъла да обхожда нея)

- Остава да си изградим LINQ заявката:
 - Избираме име на итератор. Ще търсим в списък са действия, така че един ред е подходящо да се казва `activity`:
`from activity`
 - Ще търсим в масива `currentSessionActivities`:
`from activity in currentSessionActivities`
 - Искане редовете, в които се съдържа филтъра:
`from activity in currentSessionActivities`
`where activity.Contains(filter)`
 - От получените редове искаме: целите редове:
`from activity in currentSessionActivities`
`where activity.Contains(filter)`
`select activity`
 - Получения резултат искаме във форма на списък:
`(from activity in currentSessionActivities`
`where activity.Contains(filter)`
`select activity).ToList();`
- Остава да промените извикването на функцията `Logger.GetCurrentSessionActivities()` в класа `Program` така че да позволите на администратора да въвежда филтър.

- Ако сме убедени, че резултатът от заявката ще върне само един обект, може да използваме First.
(Може да използваме First и ако искаме само първия елемент от множество резултати, естествено)

```
string tool =  
    (from t in tools  
     where t.Contains("Table")  
     select t).First();
```

Tablesaw

```
string tool =  
    (from t in tools  
     where t.Length == 8  
     select t).First();
```

Tablesaw

```
string tool =  
    (from t in tools  
     where t.Contains("Table") && t.Length == 8  
     select t).First();
```

Tablesaw

Задача: Променете функцията `UserData.IsUserPassCorrect` да извършва търсенето по списъка с потребители с LINQ заявка.

Общи задачи (пример за работа в час):

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

Задача: Променете класа **Student**

- Добавете към класа **Student** публични полета от подходящ тип и име, които да съхраняват:
 - Дата на последна заверка на семестър
 - Дата на последно плащане на семестриална такса

Какви типове са подходящи за полетата?

Каква е подходяща видимост на полетата?

Има ли нужда от специална реализация (конструктори, методи)?

Задача: Променете класа **StudentData**:

- Променете свойство **TestStudent** на **TestStudents**
- **TestStudents** да е списък с обекти от тип **Student**
- Добавете код който да добавя с писъка **TestStudents** няколко попълнени примерни обекта.
- Добавете функция **IsThereStudent()**, която да връща данните за студент по подаден факултетен номер. Реализирайте търсенето с LINQ заявка.

Какъв тип е подходящо да връща функцията?

Какви параметри са нужни на функцията?

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.