Обработка на колекция от данни

Тема 2

1. Дефинира понятието сорс-контрол система. Изброява команди за работа със сорс-контрол системите. Различава централизирани и децентрализирани сорс-контрол системи.

Сорс контрол - система за контрол на версиите, като например **SVN, GIT, GITHUB**
Контрол на версиите (Version Control) е еквивалент на управление на софтуерни конфигурации (Software Configuration Management / SCM). Това е една от дисциплините в софтуерното инженерство.

- Съдържа техники, практики и инструменти за работа със споделени файлове и програмен код
- Има механизми за управление, контрол и проследяване на промените
- Дефинира процеса на управление на промените
- Описва какво се е случило в проекта с течение на времето
- Разрешава конфликтите, възникнали при промените

Използваме системите за контрол на версиите за улесняване на работата в екип, като:

- Кодът се съхранява в централно хранилище
- Води се опис на всички промени в проекта
- Лесно се разрешават конфликтите, възникнали при сливане на промени

Git е разпределена система за контрол на програмния код (source-control system)

- Най-популярната в света (към момента)
- Свободна, софтуер с отворен код

Работи с локални и отдалечени хранилища, използва **Git bash**, който е с команден интерфейс.

Git команди:

- Клониране на съществуващо Git хранилище
  git clone [отдалечен url]
- Изтегляне и сливане на промени от отдалечено хранилище
  git pull
- Подготовка (добавяне / избор) на файлове за запис
  git add [файл] ("git add ." добавя всичко)
- Предаване (commit) към локалното хранилище
  git commit –m "[вашето съобщение]"

Системата за контрол на версиите се развива през годините, като в началото ѝ стои **локалната система за контрол на версиите** (като RCS), при която файловете са били на локалния компютър на разработчика и само той е работил с тях.

Централизирана система за контрол на версиите (CVS, Subversion), която позволява много потребители да работят по един проект. Проектът се намира на един централен компютър, с който те осъществяват връзка.

Децентрализирана система за контрол на версиите (Git), която решава най-големия недостатък на другите две – в случай на повреда в централния или локалния компютър, всички данни на проекта могат да бъдат загубени, тъй като се намират на едно физическо място. **При децентрализираната система за контрол на версиите проектът се сваля и копира локално на компютъра на всеки участващ потребител, като така се създава бекъп.** В случай на необходимост, проектът може да се възстанови от локалното копие на някой от потребителите.

2. Демонстрира откриване и отстраняване на проблем в програма с помощта на дебъгера.

Visual Studio има вграден дебъгер . Той ни предлага :

- Стопери (breakpoints)
- Възможност да следим изпълнението на кода
- Средство за наблюдение на променливите по време на изпълнението на програмата

Използване на дебъгера във Visual Studio

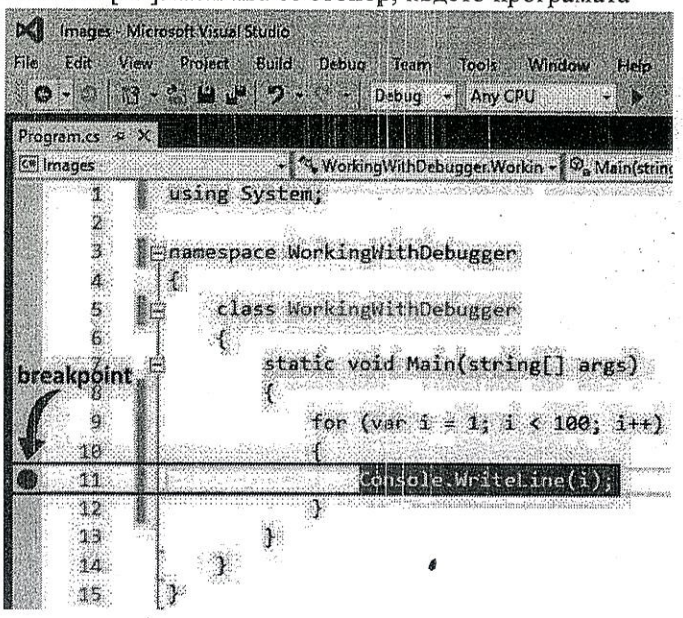
- Активиране на стопер: [F9]
- Стартиране с дебъгер: [F5]
- Проследяване на кода: [F10] / [F11]
- Използване на Locals / Watches
- Условни стопери

Visual Studio ни предоставя **вграден дебъгер** (debugger), чрез който можем да поставяме **точки на прекъсване** (или breakpoints), на избрани от нас места. При среща на **стопер** (breakpoint), програмата **спира изпълнението** си и позволява **постъпково изпълнение** на останалите редове. Дебъгването ни дава възможност да **вникнем в детайлите на програмата** и да видим къде точно възникват грешките и каква е причината за това.

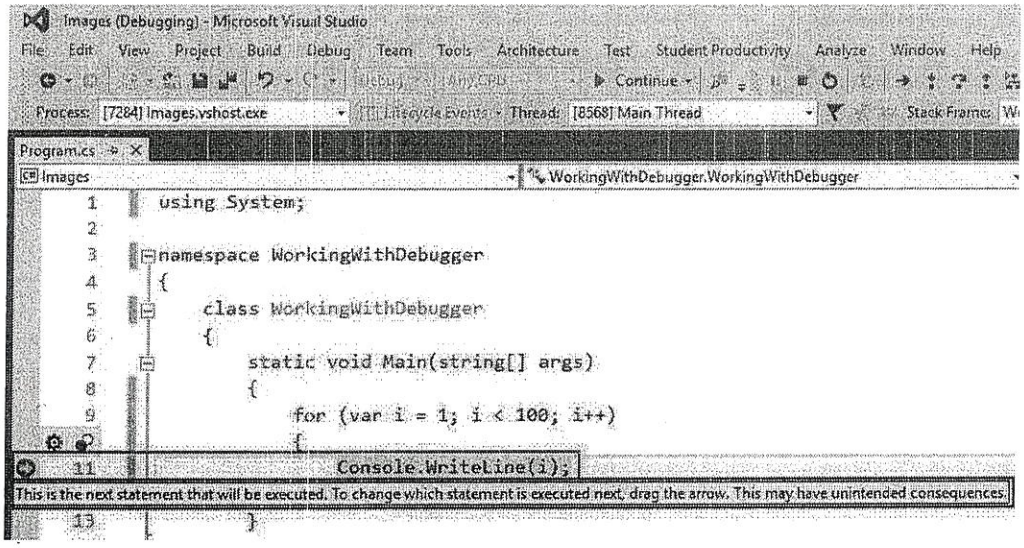
За да демонстрираме работа с дебъгера ще използваме следната програма

```
static void Main(string[] args)
{
  for (int i = 0; i < 100; i++)
  {
    Console.WriteLine(i);
  }
}
```

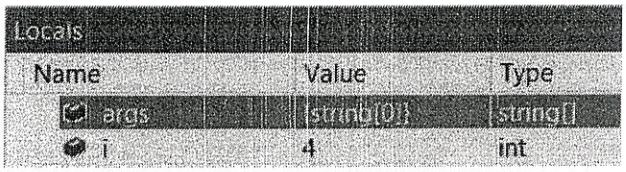
Ще сложим **стопер** (breakpoint) на функцията **Console.WriteLine(…)**. За целта трябва да преместим курсора на реда, който печата на конзолата, и да натиснем **[F9]**. Появява се **стопер**, където програмата ще **спре** изпълнението си:



За да стартираме **програмата в режим на дебъгване**, избираме **[Debug] -> [Start Debugging]** или натискаме **[F5]**. След стартиране на програмата виждаме, че тя **спира изпълнението си** на ред 11, където сложихме стопера (breakpoint). Кодът на текущия ред се **оцветява с жълт цвят** и можем да го **изпълняваме постъпково**. За да преминем на **следващ ред** използваме клавиш **[F10]**. **Забелязваме, че кодът на текущия ред все още не е изпълнен. Изпълнява се, когато преминем на следващия ред**:



От прозореца **Locals** можем да наблюдаваме **промените по локалните променливи**. За да отворите прозореца изберете [**Debug**] -> [**Windows**] -> [**Locals**].



3. Описва типове данни. Дефинира понятието обект. Посочва видове бройни системи. Преобразува числа от една бройна система в друга и изчислява изрази с тях.

Целочислени типове данни:

- **Съдържат числа**
  - Имат определени диапазони
  - Могат да пазят или да не пазят знак

Пример с целочислен тип:

```
byte centuries = 20;    // Много малко число (до 255)
ushort years = 2000;    // Малко число (до 32767)
uint days = 730484;     // Голямо число (до 4.3 млрд.)
ulong hours = 17531616; // Много голямо число (до 18.4*10^18)
Console.WriteLine(
  "{0} centuries = {1} years = {2} days = {3} hours.",
  centuries, years, days, hours);
```

Реални типове данни:

Типовете с плаваща запетая:

- Съдържат реални числа, например: 1.25, -0.38
- Имат диапазон и точност според използваната памет
- Понякога се наблюдават аномалии при изчисления
- Могат да пазят много малки и много големи стойности.

7

Типовете с плаваща запетая са:

- float
  - 32-битов, точност 7 знака след запетаята
- double
  - 64-бита, точност от 15-16 знака след запетаята
- Стойността по подразбиране е:
  - 0.0F за тип **float**
  - 0.0D за тип **double**

Реален тип с десетична точност:

- Има специален реален тип с десетична точност в C#:
  - **decimal**
    - 128-битов, с точност до 28-29 знака
  - Използва се за финансови изчисления
  - Почти няма грешки при закръгляне
  - Почти няма загуба на точност
- Стойността по подразбиране за **decimal** е:
  - **0.0M** (**M** е наставката за десетичните числа)

Преобразуване на типове:

Променливите съдържат стойности от даден тип

- Типът може да се промени (преобразува) към друг тип
  - ***Скрито преобразуване*** на тип (без загуби): променлива от по-голям тип (пр. **double**) взема по-малка стойност (пр. **float**)

```
float heightInMeters = 1.74f;
double maxHeight = heightInMeters; //Скрито преобразуване
```

  - ***Явно преобразуване*** (със загуба) – може да загубим точност:

```
double size = 3.14;
int intSize = (int) size; //Явно преобразуване → 3
```

Обектен тип в C#

- Специален тип – родител на всички други типове в .NET
- Задава се чрез object ключова дума
- Може да приема стойности, от който и да е тип
- **Референтен тип** – съдържа указател към област в паметта, на която се съхранява неговата стойност

Бройните системи (**numeral systems**) са начин за представяне (записване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна **азбука**. Освен азбука, всяка бройна система има и **основа**. Бройните системи биват позиционни и непозиционни.

- ***Десетична бройна система***

Числата представени в **десетична бройна система** (**decimal numeral system**), се задават в първичен вид, т.е. вид удобен за възприемане от човека. **Представяни, чрез 10 цифри:**

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Всяка позиция представлява умножение по 10:

За пример ще вземем числото 95031, което в десетична бройна система се представя като:

$95031 = (9\times10^4) + (5\times10^3) + (0\times10^2) + (3\times10^1) + (1\times10^0)$

- *Двоична бройна система*

Двоичните числа се преставят като последователност от битове

Най-малката единица информация – 0 или 1

Битовете лесно се представят в електрониката

За представянето на двоичните числа, се използва двоичната бройна система, която има за основа числото 2.

- Представя се с 2 цифри: 0 and 1

Всяка позиция **преставлява умножение по 2**:

101_b $= 1*2^2 + 0*2^1 + 1*2^0 = 100_b + 1_b =$

$= 4 + 1 = 5$

110_b $= 1*2^2 + 1*2^1 + 0*2^0 = 100_b + 10_b =$

$= 4 + 2 = 6$

- ***Шестнадесетична бройна система***

Шестнадесетични числа (**основа 16**)

Представяни чрез 16 цифри:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F

Преобразуване от десетична в двоична бройна система

- Делим на две и прибавяме в обратен ред остатъците:

500/2 = 250 (0)
250/2 = 125 (0)
125/2 = 62 (1)
62/2 = 31 (0)
31/2 = 15 (1)
15/2 = 7 (1)
7/2 = 3 (1)
3/2 = 1 (1)
1/2 = 0 (1)

$\longrightarrow \quad 500_d = 111110100_b$

Преобразуване на числа от шестнадесетична към десетична БС

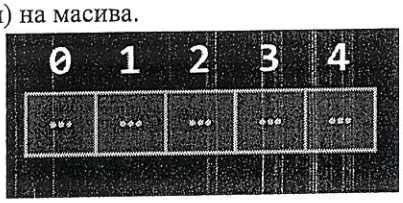
- Умножаваме всяка цифра по основата на съответния степенен показател:

$FF_{hex} = 15*16^1 + 15*16^0 = 240 + 15 = 255_d$

4. Дефинира понятието едномерен и многомерен масив. Описва декларирането на масиви. Илюстрира графично едномерни и многомерни масиви. Разработва алгоритми върху масиви.

В програмирането, масивът е множество от елементи

- Елементите са номерирани от 0 до Length-1
- Елементите са от същия тип (намример integers – цели числа)
- Масивите имат постоянен размер(дължина) (Array.Length) – не може да бъде променяна. В Length се пази дъжината (брой елементи) на масива.



Масив от 5 елемента

- Създаване на масив от 10 цели числа:

```
int[] numbers = new int[10];
```

- Задаване на стойности на елементите на масива:

```
for (int i = 0; i < numbers.Length; i++)
    numbers[i] = 1;
```

- Достъп до елементите на масива се осъществява по индекс. Операторът [] дава достъп до елементите по index:

```
numbers[5] = numbers[2] + numbers[7];
```

- **Въвеждане на масиви от конзолата -** чрез for цикъл или **String.Split()**
  - Първо, въвеждаме броя на елементите length на масива:
    int n = int.Parse(Console.ReadLine());
  - После създаваме масив с **n** на брой **елементи** и **ги** въвеждаме :

```
int[] arr = new int[n];
for (int i = 0; i < n; i++)
{
 arr[i] = int.Parse(Console.ReadLine());
}
```

- **Въвеждане стойностите на масива на един ред**
  - Стойностите на масив могат да бъдат въведени на един ред, разделени с интервал:

```
string values = Console.ReadLine();
string[] items = values.Split(' ');
int[] arr = new int[items.Length];
for (int i = 0; i < items.Length; i++)
 arr[i] = int.Parse(items[i]);
```

.Split(' ') разделя по интервал string и го записва в масив string[]

- По кратък запис, чрез функционално програмиране:

```
using System.Linq;
…
var inputLine = Console.ReadLine().Split(' '). Select(int.Parse).ToArray();
```

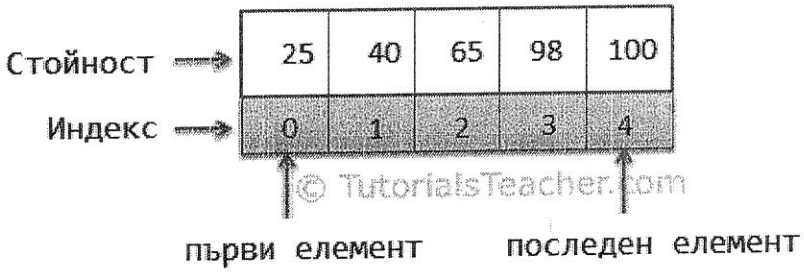
- **Извеждане на масив на конзолата:**
  - За извеждане на елементите на масив може да се ползва цикъл **for**
  - Разделяне на елементите с **интервал** или **нов ред**

```
string[] arr = {"one", "two", "three", "four", "five"};
```

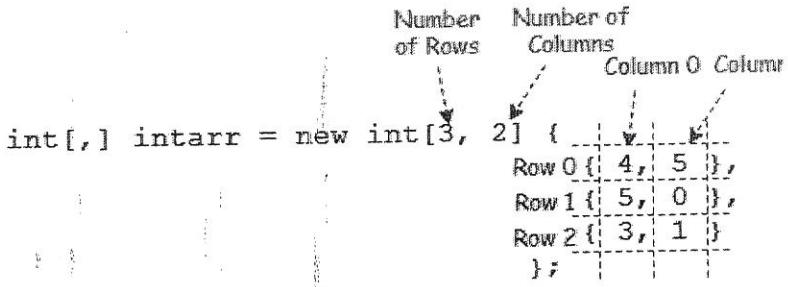
```
// преминаваме през всички елементи на масива
for (int index = 0; index < arr.Length; index++)
{
 // Извеждаме всеки елемент на нов ред
 Console.WriteLine("arr[{0}] = {1}", index, arr[index]);
}
```

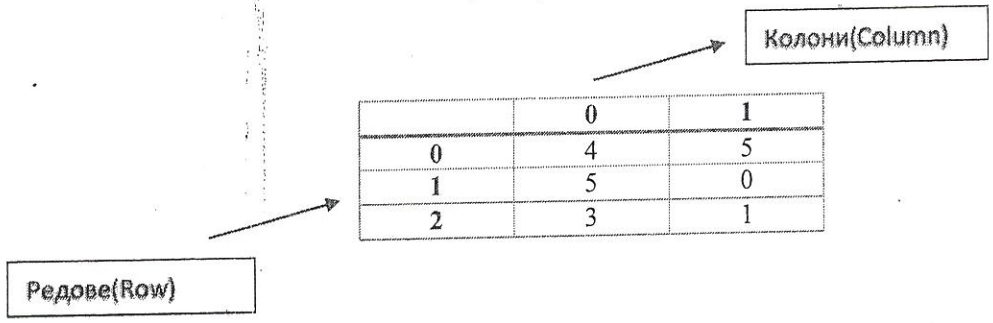
- **Илюстрира графично едномерни и многомерни масиви.**
  - **Едномерен масив**

```
int[] intArr = { 25, 40, 65, 98, 100 };
```



  - **Многомерен масив**





	0	1
0	4	5
1	5	0
2	3	1

5. Дефинира понятието списък. Описва декларирането на списъци. Дава пример за основните операции със списъци. Прави заключения за предимствата и недостатъците при употребата на списъци спрямо масиви.

Списъкът съдържа поредица от елементи (като масив, но с променлива дължина). Може да добавяме / трием / вмъкваме елементи по време на работата на програмата. C# списъкът е индексиран. Точно както при масив, можете да извлечете елемент, като предоставите неговия индекс. C# списъкът е променлив. C# списъкът е **променлив** . Това означава, че можете лесно да добавяте или премахвате елементи, когато е необходимо. Ще видим как да го направим след минута.

Създаване (декларация) на списък в C# чрез:

- Използване на конструктора без параметри и добавяне на елементи по-късно

```
var n = new List<int>();
n.Add(15);
n.Add(30);
n.Add(42);
```

- Използване на инициализатор на колекция

```
var numbers = new List<int>(new int[] { 10, 20, 30, 50} );
```

- Използване на LINQ метода „ToList()“.

Основните операции със списъци

- Добавяне на няколко елемента наведнъж. За това служи методът AddRange:

```
List<int> myList = new List<int> { 1, 2 ,3};

myList.AddRange(нов [] {4,5,6});
```

- Премахване на елементи. класът List<T> е променлив списък. Възможно е не само да добавяте елементи към него, но и да ги премахвате. Осъществява се с помощта на четири метода: Remove, RemoveAt, RemoveAll и RemoveRange.
    - **Remove(element) –** премахва първото срещане на елемент (връща **true / false**)
    - **RemoveAt(index) –** премахва елемент по неговия **индекс**
    - **RemoveAll(delegate) –** делегат, който определя условията на елементите за премахване

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
numbers.RemoveAll(x => x > 5); // премахване на числата, по-големи от 5
Console.WriteLine(string.Join(", ", numbers)); // ще отпечата "1, 2, 3, 4, 5"
```

- Вмъкване, сортиране и търсене в списък:
    - **Insert(index, element)** – вмъква елемент на зададената позиция
    - **Contains(element) –** определя дали елемента се съдържа в списъка
    - **Sort() –** сортира във възходящ ред

Предимствата и недостатъците при употребата на списъци спрямо масиви.

1. **Използвайте** списък, а не масив, когато не знаете от самото начало колко артикула ще съдържа вашата колекция.**Размерът на масива трябва да е известен, преди да инициализираме елементите му. Следователно, ако искаме да добавим елементи, надхвърлящи неговия размер, той няма да се компилира. Размерът на списъка може да бъде увеличен след деклариране.**

2. Вмъкването и изтриването са доста трудни в масив, защото се съхраняват в последователни места в паметта. За да вмъкнете и изтриете елемент, трябва да извършите операция по преместване. В тези случаи е по удобно да се използва списък, тъй като има полезни методи за вмъкване и изтриване на елементи.
3. Списъците заемат повече памет от масивите.

Извод:
Ако имате колекция с фиксиран размер, тогава би било много по-добре да използвате масиви. Например, имате низов масив, наречен names с имената на 5 души. От друга страна, ако вашата колекция трябва да бъде динамична, използвайте списъци. Например създавате списък, но ще го попълните по-късно. В този случай списъците са по-полезни. Така че можем да кажем, че както масивите, така и списъците са наистина полезни, когато се използват правилно.

6. Дефинира понятието символен низ. Описва деклариране на символен низ. Посочва методи за работа със символни низове от изучаван език за програмиране. Разработва програми за алгоритми за обработка на текст чрез операции за текстови низове (извличане на подниз, замяна на низ и др.).

Символният низ е последователност от символи, записана на даден адрес в паметта. Класът System.String позволява обработка на символни низове в C#. За декларация на низовете ще продължим да използваме служебната дума string, която е псевдоним (alias) в C# на класа System.String от .NET Framework. Работата със string ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции.

Декларация на символен низ:

```
string s = "Hello, C#";
```

Декларирахме променливата greeting от тип string, която има съдържание "Hello, C#". Представянето на съдържанието в символния низ изглежда по подобен начин.

H	e	l	l	o	,		C	#

Можем да **инициализираме** променливи по **3 начина**:

1. Чрез задаване на низов литерал.
2. Чрез присвояване стойността от друг символен низ.
3. Чрез предаване стойността на операция, връщаща символен низ.

Методи за работа със символни низове:

- **Сравняване за еднаквост** - Ако условието изисква да сравним два символни низа и да установим дали стойностите им са еднакви или не, удобен метод е методът Equals(…), който работи еквивалентно на оператора ==.
- **Сравнение на низове по азбучен ред** - Ако искаме да сравним две думи и да получим информация коя от тях е преди другата, според азбучния ред на буквите в нея, на помощ идва методът CompareTo(…). Той ни дава възможност да сравняваме стой-ностите на два символни низа и да установяваме лексикографската им наредба.
- **Преминаване към главни и малки букви** – Ако искаме да променим съдържанието на символен низ, така че всички символи в него да бъдат само с главни или малки букви се използват двата метода ToLower(…) и ToUpper(…)
- **Търсене на низ в друг низ** - Методите IndexOf(…) и LastIndexOf(…) претърсват съдържанието на текстова последователност, но в различна посока. Когато се интересуваме от първото срещнато съвпадение, използваме

10

IndexOf(…). Ако искаме да претърсваме низа от неговия край (например за откриване на последната точка в името на даден файл или последната наклонена черта в URL адрес), ползваме LastIndexOf(…).

- **Извличане на част от низ -** Методът Substring(startIndex, length) се използва да извлечем дадена част от низ (подниз) по зададени начална позиция в текста и дължина.

7. Дефинира понятието речник. Описва устройството на речник (хеш-таблица). Посочва методи за работа с речник. Решава задачи с използването на подходящи методи върху речник. Различава ключ и стой.

- **Речника** е асоциативните масив, чиито индекси са ключове. Ключовете могат да бъдат думи или пък реални числа, за разлика от индексите на обикновения масив. Речника съдържат информация в двойки **{ключ → стойност}.**

  Създаване на речник става с помощта на клас **Dictionary<TKey, TValue>**:

  Dictionary dictionary_name = new Dictionary();

  Ако искате да добавите елементи във вашия речник може да използвате:
  - Метод Add(key,value)
  - Чрез използване на индекси
  - Чрез използване на foreach цикъла за да достъпите key/value част от речника.

  Методи за работа с речник:

  - Изтриване на елемент от речник
    - Clear() - Този метод премахва всички ключове и стойности
    - Remove(key) - Този метод премахва стойността с посочения ключ от речника
  - Проверка за наличност на елемент в речник
    - ContainsKey()- Този метод се използва за проверка дали речника съдържа указания ключ.
    - ContainsValue()- Този метод се използва за проверка дали речника съдържа конкретна стойност.
    - **TryGetValue() –** проверяваме дали даден ключ съществува в речника и отпечатва стойността му

  Свойства за работа с речници:

- **Count** – пази броя на двойките от ключ-стойност
- **Keys** – съдържа уникалните ключове
- **Values** – съдържа всички стойности

Пример:

```
static public void Main() {

        // Създаване на речник
        Dictionary<int, string> My_dict =
                    new Dictionary<int, string>();

          // добавяне на ключ и стойност с метод Add()
          My_dict.Add(1123, "Welcome");
          My_dict.Add(1124, "to");
          My_dict.Add(1125, "GeeksforGeeks");
```

```
// Отпечатване на елементите преди Remove()
        foreach(KeyValuePair<int, string> ele in My_dict)
        {
            Console.WriteLine("{0} and {1}",
                        ele.Key, ele.Value);
        }
        Console.WriteLine();

        My_dict.Remove(1123);

        //Отпечатване след Remove() метод
        foreach(KeyValuePair<int, string> ele in My_dict)
        {
            Console.WriteLine("{0} and {1}",
                        ele.Key, ele.Value);
        }
        Console.WriteLine();
}
```