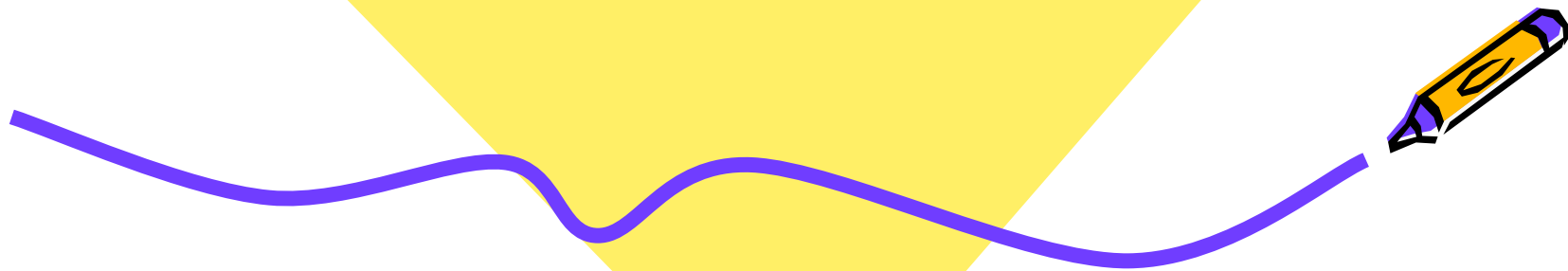


# Лекция 8. Коллекции с общо предназначение в .Net Framework



# Колекция ?

- Колекциите ("класове - контейнери", "контейнер-класове" или само "контейнери" - Наков) са абстрактни типове данни (ADT) и могат да бъдат имплементирани по различен начин, например: чрез масив, чрез свързан списък, чрез различни видове дървета, чрез пирамида, чрез хеш-таблица и т. н.
- Различни видове колекции могат да бъдат избирани за съхраняване на данни, но изборът на конкретна колекция зависи от планирането на редица фактори, например **от планираният начин за управление или достъп до елементите.**



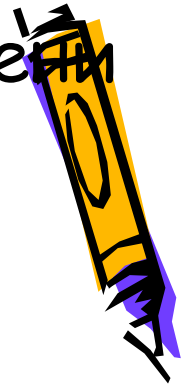
- Така например, в една списъчна колекция - `ArrayList`, добавянето на елемент в началото или в средата на колекцията е по-бързо, в сравнение с това в масив. Освен това, тази колекция с променлив размер, а е известно че има колекции, които са с фиксиран размер (например масивите).
- Колекциите могат да бъдат само за четене или да позволяват и промени.
- .Net Framework поддържа 4 типа колекции:
  - колекции с общо предназначение (в `System.Collections`)
  - колекции със специално предназначение (в `System.Collections.Specialized` ),
  - типизирани колекции, базирани на тъй наречените `generics` (намират се в `System.Collections.Generic`). Тези колекции са въведени за пръв път в .Net Framework 2.0.



- В .NET Framework класовете, имплементиращи колекции се намират в пространството
- `System.Collections`, `System.Collections.Generic` и `System.Collection.Specialized`.
- Колекциите в `System.Collections` са колекции с общо предназначение, елементите им са от тип `object`, ето защо могат да се използват за съхраняване на всеки тип данни. Има едно изключение - една единствена колекция в `System.Collection (BitArray)` прави изключение - служи за съхраняване на групи от битове и поддържа специфичен набор от операции над битове.



- В `System.Collection.Specialized` са разположени колекции със специално предназначение - ориентирани са към обработка на конкретен тип данни или към обработка на данни по специален начин. Например, съществуват специализирани колекции, предназначени само за обработка на стрингове.

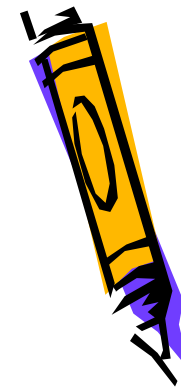
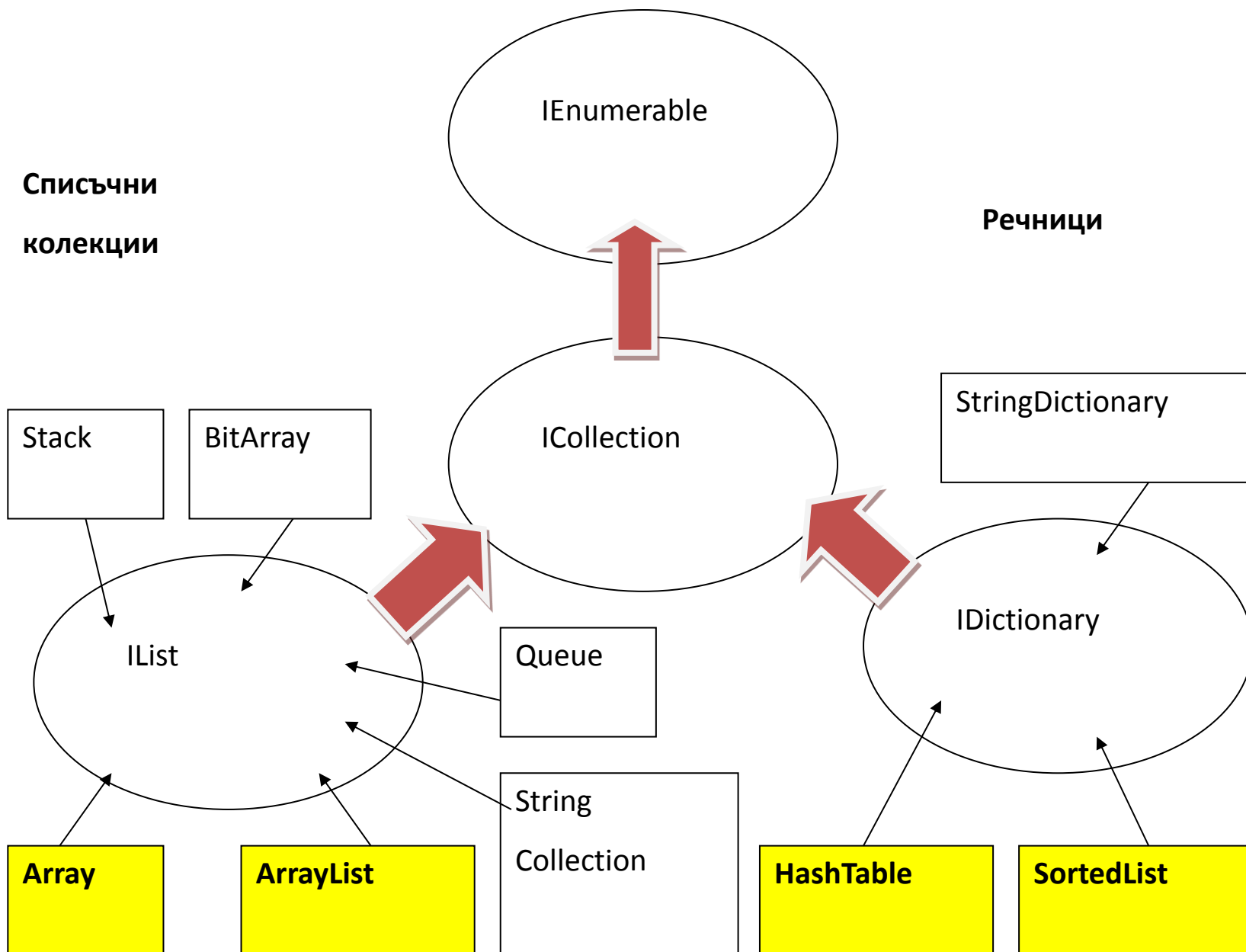


- Поддържането и използването на колекции в FCL, в това число и на масивите, се основава на интерфейси. Всички колекции в .NET Framework имплементират един или няколко от интерфейсите **IEnumerable**, **ICollection**, **IDictionary** и  **IList**.
- Нека отново припомним клас-диаграмата, изобразяваща нагледно йерархията на тези интерфейси.



Списъчни  
колекции

Речници



- Нека по-подробно разгледаме интерфейсите **IEnumerable**, **ICollection** и обясним за какво служат.

- **IEnumerable:**

- Всички колекции в .NET Framework наследяват този интерфейс.
- Той осигурява поддръжката на операцията "обхождане на всички елементи". Дефинира само един метод - методът `GetEnumerator()`, който връща изброител или итератор (инстанция на интерфейса `IEnumerator`), с който се извършва самото обхождане. Изброителят предоставя един универсален начин за обхождане на колекция, независимо от нейния тип.





```
public interface IEnumerable()  
{IEnumerator GetEnumerator();}
```

//метод GetEnumerator () връща итератор - обект от тип IEnumerator

- Интерфейс IEnumerator дефинира универсален начин за изброяване (обхождане) на обектите на колекция от какъвто и да е тип. Има 1 свойство и 2 метода:

```
public interface IEnumerator  
{bool MoveNext();
```

//преместване на една позиция напред в контейнера от елементи

```
object Current {get;}
```

//текущ елемент в контейнера. При първоначално създаване, изброителят се намира преди първия елемент.

```
void Reset(); //преместване в началото на контейнера
```



- Пример:

```
...using System.Collections;...
```

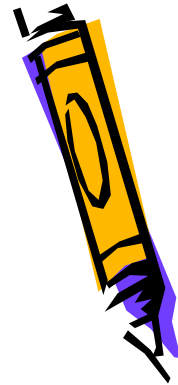
```
int[] names = { 1, 3, 5, 7 };
```

```
IEnumerator e = names.GetEnumerator();
```

```
//1. Обхождане с изброител „e“
```

```
while (e.MoveNext())
```

```
Console.WriteLine((int)e.Current);
```



- **Интерфейсът ICollection**

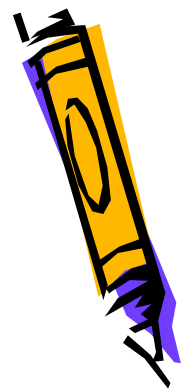
Интерфейсът **ICollection** е базов за всички колекции в .NET Framework.

Той разширява **IEnumerable** и добавя към него свойството **Count**, което връща общия брой елементи в дадена колекция.

- **Пример:**

```
ArrayList myArray = new ArrayList();...
```

```
Console.WriteLine("Текущо количество  
елементи: " + myArray.Count);
```



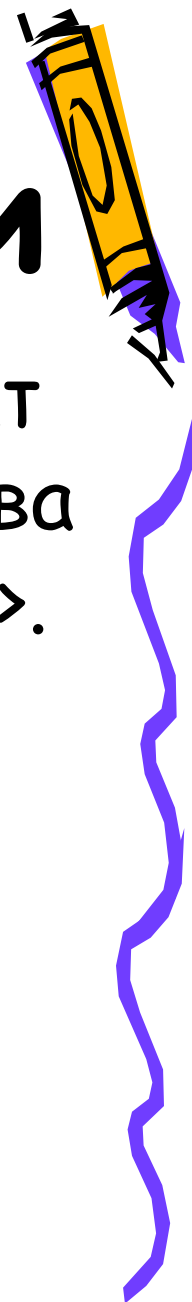
# Списъчни и речникови колекции

- Както видяхме на фигурата по-напред, колекциите в C# са два вида:
- Списъчните се характеризират с това, че имплементират интерфейса **ICollection**.  
Такива са **Array**, **ArrayList**, **Queue**, **Stack**, **BitArray** и **StringCollection**.



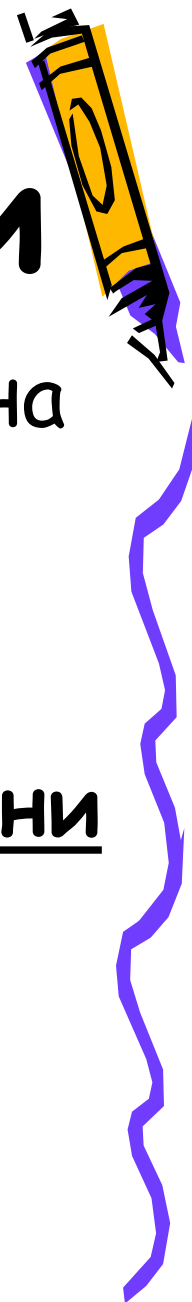
# Списъчни и речникови колекции

- Речниковите колекции имплементират интерфейса **IDictionary** - представлява колекция от двойки <ключ>-<стойност>.
- Примери за речникови колекции са класовете **Hashtable**, **SortedList** и **StringDictionary**.



# Списъчни и речникови колекции

- Изброените колекции, с изключение на `Array`, се намират в именовано пространство `System.Collections` и с изключение на `BitArray` (списък от булеви стойности) са слабо типизирани - елементите им са от тип `System.Object`.



# Списъчни и речникови колекции

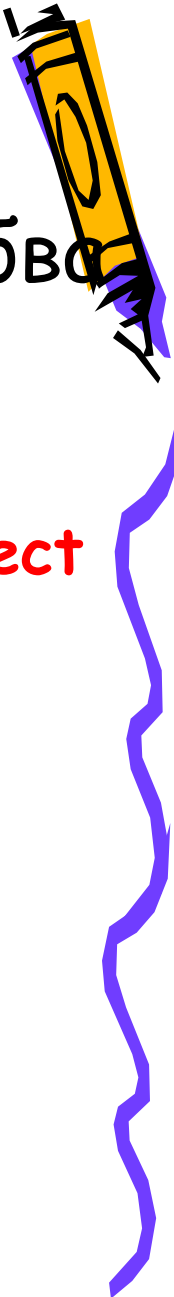
- Предимство: Слабата типизация позволява на колекциите да съдържат фактически всякакъв тип данни, защото всеки тип данни в .NET Framework наследява **System. Object**. Ето един пример:

```
ArrayList list = new ArrayList();  
list.Add("beer"); // string inherits System.Object  
string s = (string) list[0];
```



- Недостатък: За съжаление слабата типизация означава, че всеки път, при достъп до елемент от колекцията, трябва да се прави преобразуване на типове.

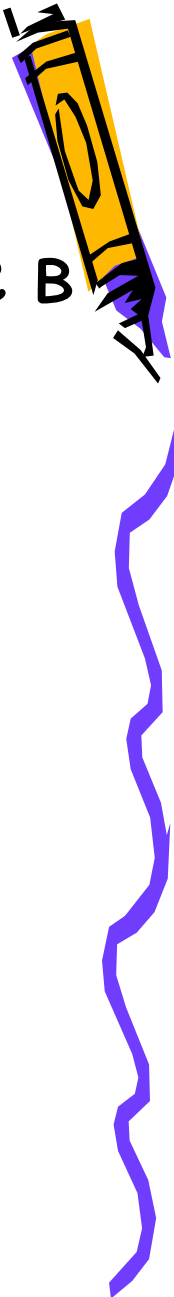
```
ArrayList list = new ArrayList();  
list.Add("beer"); // string inherits System.Object  
string s = (string) list[0];
```





- !!!наблюдава намалена производителност

- При съхранение на стойностни типове в колекции, те се преобразуват в референтни, тоест прави "опаковане" (boxing), а при достъп до елемент от колекцията е налице обратно преобразуване, от референтен към стойностен тип, съответно - "разопаковане" (unboxing).



- В .NET Framework 2.0 бяха въведени нови типизирани колекции (базирани на т. нар. *generics*).
- Те наподобяват така наречените шаблони (*templates*) в C++ - очаква се до голяма степен да решат проблема с намалената производителност от липсата на типизация.



# Списъчни колекции

## Интерфейсът IList

- предоставя формално описание на една подредена колекция от обекти и множество от средства за работа с тази колекция:
  - достъп до елементите по индекс,
  - добавяне на елемент (**Add(...)**),
  - вмъкване на елемент (**Insert(...)**),
  - търсене на елемент (**IndexOf(...)**),
  - изтриване по стойност **Remove(...)**, по индекс **RemoveAt(...)**, от-до **RemoveRange(...)**, и др.



# Списъчни колекции

Имплементациите на IList са 3 вида:

- ReadOnly
- С фиксиран размер
- С променлив размер

IList предоставя следните public свойства:

- bool IsFixedSize {get;} - връща True ако колекцията е с фиксиран размер.
- bool ReadOnly {get;} - връща True ако колекцията е ReadOnly.
- Item - установява или връща елемент, на указаното чрез индекс [] място



```
ArrayList list = new ArrayList();
```

```
list.Add("Анна");
```

```
// Добавя се елемент към края на  
ArrayList - list[0]
```

```
list.Add("Жана");
```

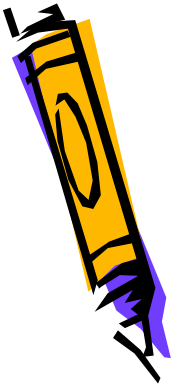
```
// Добавя се стойност за елемент list[1]
```

```
list.Add("Иван");
```

```
// Добавя се стойност за елемент list[2]
```

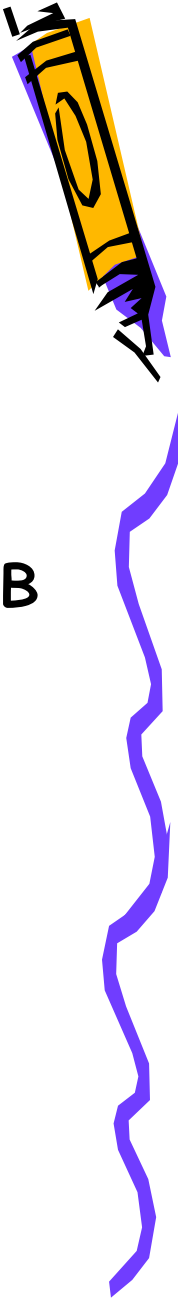
```
list[2] = "Сияна";
```

```
// установява нова стойност за елемент  
list[2]
```



# Класът ArrayList

- Основна разлика между Array и ArrayList е, че при Array се използва имплементацията на IList с фиксиран размер, докато за ArrayList - имплементацията на IList с променлив размер.



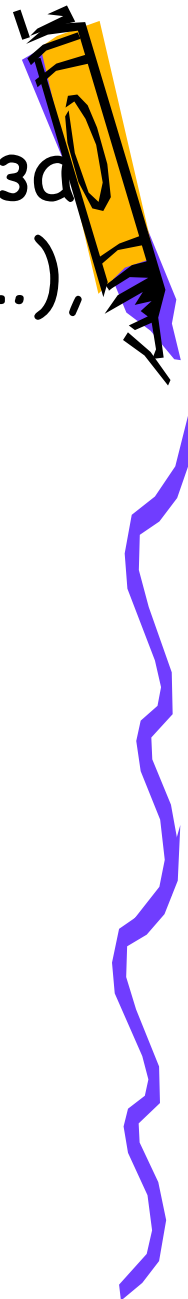
# Класът ArrayList

- Така, класът ArrayList имплементира интерфейса IList чрез масив, чийто размер се променя динамично при нужда: Всяка инстанция на този клас предварително заделя буферна памет (Capacity) за елементите (=16 елемента), които предстои да бъдат добавени. При запълване на буферната памет се заделя нова памет, като най-често капацитетът се удвоява.



# Класът ArrayList

- ArrayList има някои методи, типични за масивите (методи Sort(...), BinSearch(...), Reverse(...)). Може да се превръща в масив (метод ToArray(...)).





## Конструктори на ArrayList:

1.

```
public ArrayList();//ArrayList list = new  
    ArrayList();
```

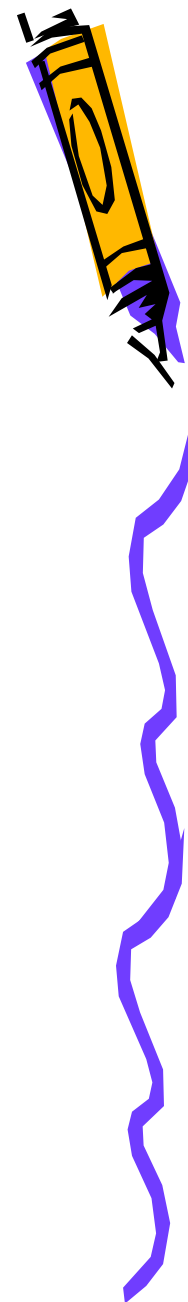
2.

```
public ArrayList(число);//указваме  
    капацитет
```

```
ArrayList list = new ArrayList(1000);
```

3.

```
public ArrayList(ICollection Coll);  
//запълваме list с елементи от Coll.
```



# Методи на ArrayList:

**int Add(object value);**

//Добавя елемент в края, връща индекс

ArrayList a = new ArrayList();

a.Add("Anna"); // Adds at the end of the ArrayList

**bool Contains(object value);**

//Търси елемент в списъка. Връща True, ако го намери

Console.WriteLine(a.Contains("Anna"));

**int IndexOf(object value);**

//Връща индекса на елемент в списъка.

Console.WriteLine(a.IndexOf ("Anna"));//0



## Методи на ArrayList:

**Insert(int index, object value);** //Вмъква на  
зададено място

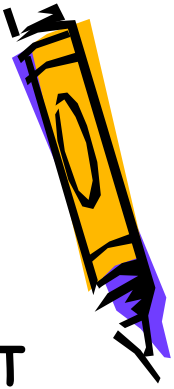
a.Insert(0,"Alina");//Елементите се преместват  
надолу

**void Remove(object value);** //Премахва  
стойност

a.Remove("Alina");

**RemoveAt(int index);** //Премахва по индекс

a.RemoveAt (1); //2-рия елемент изчезва



# Методи на ArrayList:

## RemoveRange(От - До)

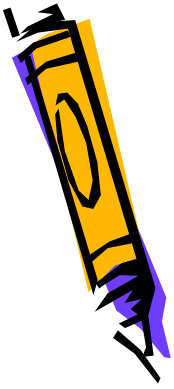
a.RemoveRange(1,10); //от 2-рия до 11-тия  
елемент изчезват

## a.Clear(); //Изчистваме данните

Console.WriteLine(list.IndexOf("Varna")); //-1

## a.TrimToSize();

// За да намалите размера на ArrayList до  
действителния брой елементи, тъй като  
автоматично се заделя място



### Пример:

```
static void Main()
{
    ArrayList list = new ArrayList();
    for (int i = 1; i <= 10; i++)
    {
        list.Add(i); // Adds i at the end of the ArrayList
    }
    list.Insert(3, 123); // Inserts 123 element number 3
    list.RemoveAt(7); // Removes element with the index 7
    list.Remove(2); // Removes element with value 2
    list[1] = 500; // Changes element with index 1
    list.Sort(); // Sorting in ascending order
    int[] arr = (int[])list.ToArray(typeof(int));
    foreach(int i in arr)
    {
        Console.Write("{0} ", i);
    }
    Console.WriteLine();
    // Result: 1 4 5 6 8 9 10 123 500
}
```



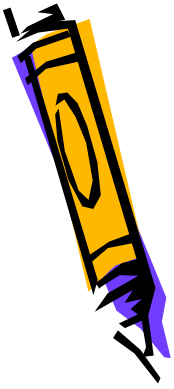
- Други списъчни колекции
- Освен **ArrayList** в .NET Framework стандартно са имплементирани и някои други списъчни колекции, като опашки и стекове.
- **Queue** - Опашката представлява колекция с поведение от вида "first-in, first-out (FIFO)" и е реализирана чрез цикличен масив. Класическа аналогия за тази структура е опашката за билети. Този, който първи се е наредил на опашката, ще може първи да си купи билет. Характерни за класа **Queue** са двата метода **Enqueue(...)** и **Dequeue(...)**, служещи съответно за добавяне и изваждане на елемент от опашката. **Enqueue(...)** добавя елемент в края на опашката, а **Dequeue(...)** изважда елемент от началото ѝ.
- **Stack** - За разлика от опашката, стекът представлява структура с поведение от вида "last-in, first-out (LIFO)", която се реализира чрез масив. Той работи на принципа "който е влязъл последен в стека, стои най-отгоре" - точно като колона от чинии, поставени една върху друга. Основните методи за добавяне и премахване на елемент от стека са **Push(...)** и **Pop()**. **Push(...)** добавя елемент към върха на стека, а **Pop()** връща елемента от върха на стека, като го премахва. Класът **Stack** съдържа и още метода **Peek()**, който връща елемента от върха на стека, но без да го премахва.19



```
Queue queue = new Queue();  
queue.Enqueue("1. Ivan");  
queue.Enqueue("2. Dragan");  
queue.Enqueue("3. Petkan");
```

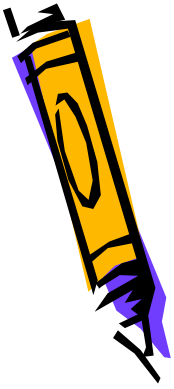
```
while (queue.Count > 0)  
{  
    string computer = (string)queue.Dequeue();  
    Console.Write("{0} ", computer);  
}  
Console.WriteLine();
```

// Result: 1. Ivan 2. Dragan 3. Petkan



```
Stack stack = new Stack();  
stack.Push("1. Ivan");  
stack.Push("2. Dragan");  
stack.Push("3. Petkan");
```

```
while (stack.Count > 0)  
{  
    string computer = (string)stack.Pop();  
    Console.Write("{0} ", computer);  
}  
Console.WriteLine();  
// Result: 3. Petkan 2. Dragan 1. Ivan
```

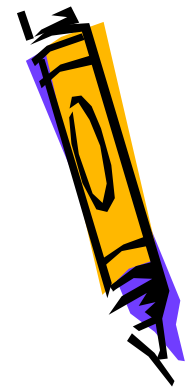




## Речникови колекции

Нека сега разгледаме и по-сложната част от средствата за работа с колекции в .NET Framework – речниковите колекции или само речници:

- **Всеки елемент** на речникова колекция представлява **двойка** от тип **ключ-стойност (key-value)**, която се съхранява в обект от тип **DictionaryEntry**.
- **Ключът** на всяка двойка трябва да е **уникален** (за което се грижим сами) и различен от **null**, а стойността, асоциирана с този ключ, може да е какъвто и да е обект, включително **null**.



## Интерфейсът IDictionary

- Интерфейсът **IDictionary** е базов за речниковите колекции, тоест този интерфейс се наследява от всички колекции, асоцииращи <ключове> и <стойности>.
- Интерфейсът **IDictionary** позволява съдържаните в колекцията ключове да се изброяват (enumerated) тоест обхождат, но не ги сортира по какъвто и да е признак.

**IDictionary** поддържа операциите:

- добавяне на нова двойка ключ-стойност (**Add(...)**),
- търсене на стойност по ключ (индексатор),
- премахване на двойка по ключ (**Remove(...)**),
- извличане на всички ключове (**Keys**),
- извличане на всички стойности (**Values**).
- Имплементациите на **IDictionary** биват няколко вида:
  - само за четене (Read-only) - само за четене, не се позволява промяна на елементите им)
  - с фиксиран размер (Fixed-size) - не се позволява добавяне и премахване на елементи, но е позволена промяната на вече съществуващи елементи
  - с променлив размер (Variable-size) - позволено добавяне, премахване и промяна на елементи.



## Особености на речниците

Нека: `Hashtable table=new Hashtable();`

1. **Имат собствен изброител.** При изброяване (обхождане) се използва собствен изброител, различен от този при списъците (тоест не се използва изброител - обект от тип `IEnumerator` интерфейс), а се използва обект от тип - интерфейс **`IDictionaryEnumerator`**. Интерфейсът `IDictionaryEnumerator` наследява интерфейс `IEnumerator`, добавяйки му методи за връщане на „key“ и „value“ на обекта.

**`public interface IDictionaryEnumerator : IEnumerator`**

2. Имат свойства:

- `ICollection Keys(get;)` // Свойство `Keys` връща колекция от ключове
- `ICollection Values(get;)` // Свойство `Values` връща колекция от стойности
- Пример: `ICollection k = table.Keys;`  
`ICollection v = table.Values;`

3. Индексатор: `this[object key]{get;set;};`

`table["1254"] = "Varna";`

`Console.WriteLine(table["1254"]);`

- Индексаторът извлича или установява стойности на елемента със засаден ключ, тоест достъпът до елементите на речник е по ключ. Ако такъв ключ съществува, то със `set` презаписваме стойността, Ако не съществува - става добавяне на такъв елемент.

Естествено добавяне може да стане и с `Add`: `table.Add("1254", "Varna");`



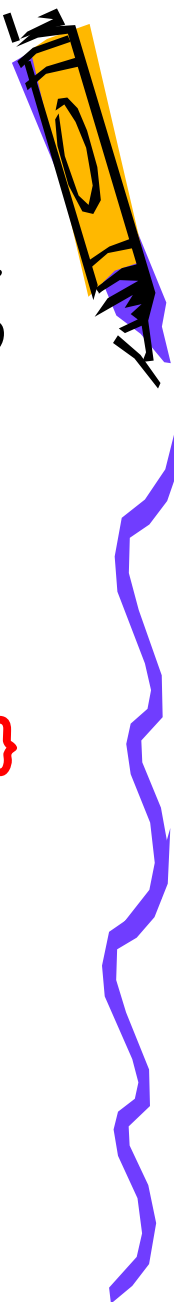
- !!! Разликата при добавяне с Add е, че ако ключът съществува вече, то ще се генерира изключение.
- Програмно елементите на един речник са достъпни като структура struct DictionaryEntry, с 2 Read-Write свойства key и value от тип object. Така обхождането на един речник може да стане по един от следните начини:

```
Hashtable table = new Hashtable();  
foreach (DictionaryEntry d in table)  
    { Console.WriteLine("{0}\t{1}", d.Key, d.Value);}
```

- Или:

```
IDictionaryEnumerator e = table.GetEnumerator();  
while(e.MoveNext())  
    Console.WriteLine("{0}\t{1}", (String)e.Key,  
        (String)e.Value);
```

// тоест има кастване



- !!! Разликата при добавяне с Add е, че ако ключът съществува вече, то ще се генерира изключение.
- Програмно елементите на един речник са достъпни като структура `struct DictionaryEntry`, с 2 Read-Write свойства `key` и `value` от тип `object`. Така обхождането на един речник може да стане по един от следните начини:

```
Hashtable table = new Hashtable();  
foreach (DictionaryEntry d in table)  
    { Console.WriteLine("{0}\t{1}", d.Key, d.Value);}
```

- Или:

```
IDictionaryEnumerator e = table.GetEnumerator();  
    while(e.MoveNext())  
Console.WriteLine("{0}\t{1}", (String)e.Key,  
    (String)e.Value);
```

- //тоест има кастване



## Клас Hashtable

- **Hashtable** е колекция с променлив размер в общия случай, наследяваща интерфейс **IDictionary**, което означава, че всеки елемент е двойка ключ-стойност, като всеки ключ трябва да е уникален (грижим се сами за това) и различен от **null**.
- Този клас (**Hashtable**) представлява имплементация на структурата от данни "хеш таблица" – речникова колекция, елементите на която се разполагат в специално заделена памет в зависимост от хеш кода на ключа на всяка от тях.
- Характерното за този речник е използването на **хеш кода на ключа**. Обектите, които се използват за ключове в хеш-таблица, наследяват методите **GetHashCode()** и **Equals(...)**. Тази структура от данни ("хеш-таблица") всъщност не може да работи без функция за пресмятане на хеш-код за създаване и съхраняване на ключове и без функция за сравнение на ключове. При това ключовете, които се считат за еднакви, задължително трябва да имат еднакъв хеш-код.



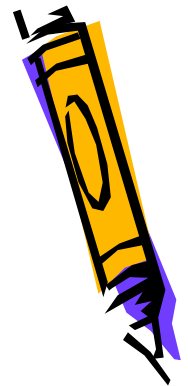
## Клас Hashtable

- Тъй като принципно всеки клас (в това число и клас **Hashtable**) наследява **System.Object**, то той автоматично наследява и предефинирана имплементация на **Equals(...)**. За съжаление, в общия случай тази имплементация се реализира чрез сравнение за съвпадение на референциите на двата обекта. Това генерално погледнато е грешен начин за сравнение и затова се налага да се имплементират специфични реализации за създадените от нас класове. Имплементацията на **Equals(...)** трябва да връща винаги един и същ резултат, когато се вика с едни и същи параметри.
- Класът **Hashtable** имплементира интерфейса **IDictionary**, а той от своя страна знаем че наследява **IEnumerable**. Това ни позволява да използваме оператора **foreach** за да обхождаме елементите на хеш-таблици.
- Примерът по долу демонстрира работата с класа **Hashtable** и показва как се използват основните операции, свързани с него: Демонстрират се различни варианти на добавяне, извличане, изтриване и промяна на елементи в хеш-таблица, както и обхождане на всички елементи с **foreach** и изброител.



## Клас Hashtable - пример

```
using System;
using System.Collections; //Задължително!!!
static void Main()
{
    //Създаване на обект от тип Hashtable
    Hashtable students = new Hashtable();
    //Запълване със стойности
        students.Add("Ivan", 5);        students.Add("Kalin", 4);
        students.Add("Irina", 4);       students.Add("Ina", 6);
    //Разпечатване на успеха на "Ivan"
    Console.WriteLine("Student {0}, uspeh {1}", "Ivan", students["Ivan"]);
    //Изтриване на Ivan
        students.Remove("Ivan");
    //Промяна на успеха на Kalin
        students["Kalin"] = 3;
    //Обхождане с foreach
    Console.WriteLine(" ***** ");
        foreach (DictionaryEntry st in students)
        {
            Console.WriteLine("Student {0}, uspeh {1}",
                st.Key, st.Value); }
}
```





## Клас Hashtable - пример

**//вариант 2: Обхождане с изброител, няма Current**

```
IDictionaryEnumerator e = students.GetEnumerator();  
    while (e.MoveNext())  
        Console.WriteLine("{0}\t{1}", (String)e.Key, (int)e.Value);  
//има кастване!!!
```

**//Вариант1: Търсене по ключ**

```
Console.WriteLine("\nVavedi ime na student");  
    string input=Console.ReadLine();  
    if (students.ContainsKey(input))  
        Console.WriteLine(students[input]);
```

**//Вариант 2 Търсене по ключ с foreach**

```
Console.WriteLine("\nVavedi ime na student");  
input = Console.ReadLine();  
    foreach (DictionaryEntry st in students)  
        {if ((String)st.Key==input)  
Console.WriteLine("Student {0}, uspeh {1}",st.Key, st.Value);  
    }
```




## Клас Hashtable - пример

**//вариант1: Търсене по стойност**

```
Console.WriteLine("\nVavedi uspeh na student");  
int inp = Convert.ToInt32(Console.ReadLine());  
if (students.ContainsValue(inp))  
    Console.WriteLine("е в spisaka!!!");
```

**//Вариант 2: Търсене по стойност с foreach**

```
Console.WriteLine("\nVavedi uspeh na student");  
int inp = Convert.ToInt32(Console.ReadLine());  
foreach (DictionaryEntry st in students)  
{  
    if ((int)st.Value == inp)  
        Console.WriteLine("Student {0}, uspeh {1}",st.Key, st.Value);  
}  
Console.ReadKey();  
}
```

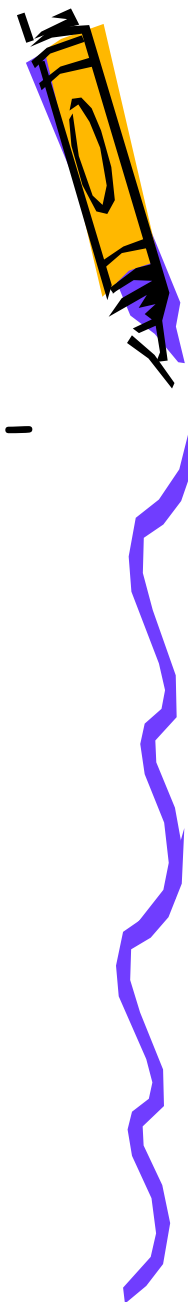


```
C:\ file:///D:/Violeta_new/C#proekti1/Con...  
Student Ivan, uspeh 5  
*****  
Student Kalin, uspeh 3  
Student Ina, uspeh 6  
Student Irina, uspeh 4  
Kalin 3  
Ina 6  
Irina 4  
  
Vavedi ime na student  
Ina  
6  
  
Vavedi ime na student  
Ina  
Student Ina, uspeh 6  
  
Vavedi uspeh na student  
4  
е в spisaka!!!  
  
Vavedi uspeh na student  
4  
Student Irina, uspeh 4  
-
```



# Клас SortedList

- Тази речникова колекция, съхранява двойки елементи ключ-стойност, но сортирани по ключ (за разлика от хеш-таблица).
- Тъй като е нужна непрекъснатата поддръжка на сортирана последователност, **SortedList** работи доста бавно от **Hashtable**, което е и основния недостатък на този речник (повечето операции имат линейна сложност).



# Клас SortedList

- Клас **SortedList**, представлява имплементация на интерфейса **IDictionary**, която прилича както на хеш-таблица, така и на масив.
- Позволява гъвкав достъп до елементите, което е и основно предимство пред **Hashtable**:
  - Индексиран достъп (като масив). Достъпът по индекс става чрез методите `GetByIndex()` и `SetByIndex`.
  - Достъп по ключ (като хеш-таблица).



# Клас SortedList

- Вътрешно SortedList е организиран като 2 масива:
  - един за ключовете и
  - един за стойностите.
- Свойство Count връща броя на двойките в SortedList.
- Основните операции с този речник ще покажем с примера по-долу:



# Клас SortedList

SortedList - пример:

- В примера класът SortedList е използван за да съхранява съответствия между студенти и успех. Понеже за ключ се използват имената, след добавянето на няколко имена в сортирания списък, те могат да бъдат извлечени след това в азбучен ред чрез просто обхождане.
- След изпълнение, примерът извежда на конзолата следното:



# Клас SortedList

```
using System.Collections; // клас SortedList е тук
```

```
static void Main()
```

```
{
```

```
//Създаване на обект от тип SortedList
```

```
    SortedList sl = new SortedList();
```

```
//Запълване със стойности
```

```
    sl.Add("Ivan", 6);
```

```
    sl.Add("Dragan", 2);
```

```
    sl.Add("Kalin", 4);
```

```
    sl.Add("Ina", 5);
```

```
//Разпечатване на успеха на "Ivan"
```

```
    Console.WriteLine("Student {0}, uspeh {1}", "Ivan", sl["Ivan"]);
```

```
//Изтриване на Ivan
```

```
    sl.Remove("Ivan");
```

```
//ОТНОВО ВЪВЕЖДАНЕ НА Ivan
```

```
    sl["Ivan"] = 3;
```



# Клас SortedList

//Промяна на успеха на Kalin

```
sl["Kalin"] = 6;
```

**//вариант 1. Обхождане с foreach**

```
Console.WriteLine("1. ***** ");
```

```
foreach (DictionaryEntry st in sl)
```

```
{    Console.WriteLine("Student {0}, uspeh {1}",  
    st.Key, st.Value);    }
```

```
Console.WriteLine("2. ***** ");
```

**//вариант 2: Обхождане с изброител, няма Current**

```
IDictionaryEnumerator e = sl.GetEnumerator();
```

```
while (e.MoveNext())
```

```
    Console.WriteLine("{0}\t{1}", (String)e.Key, (int)e.Value);
```

**//има кастване!!!**





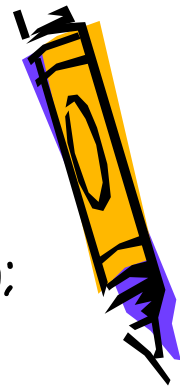
# Клас SortedList

**//вариант 3. Обхождане с използване на GetKeyList()**

```
Console.WriteLine("\n3. V spisaka sa samo slednite imena:");  
foreach (string c in sl.GetKeyList())  
{  
    Console.WriteLine("{0} ", c);  
}
```

**//вариант 4. Обхождане като масив**

```
Console.WriteLine("\n4. Imena - uspeh");  
for (int i = 0; i < sl.Count; i++)  
{  
    Console.WriteLine("{0} - {1}",  
        sl.GetKey(i), sl.GetByIndex(i));  
}
```



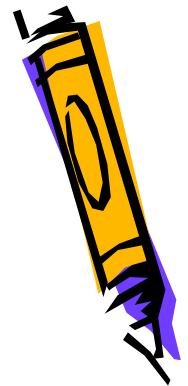
# Клас SortedList

//Вариант1: Търсене по ключ

```
Console.WriteLine("\nВведи име на student");  
string input = Console.ReadLine();  
if (sl.ContainsKey(input))  
    Console.WriteLine(sl[input]);
```

//Вариант 2 Търсене по ключ с foreach

```
Console.WriteLine("\nВведи име на student");  
input = Console.ReadLine();  
foreach (DictionaryEntry st in sl)  
{  
    if ((String)st.Key == input)  
        Console.WriteLine("Student {0}, uspeh {1}", st.Key, st.Value);  
}
```



# Клас SortedList

//вариант1: Търсене по стойност

```
Console.WriteLine("\nVavedi uspeh na student");  
int inp = Convert.ToInt32(Console.ReadLine());  
if (sl.ContainsValue(inp))  
    Console.WriteLine("е в spisaka!!!");
```

//Вариант 2: Търсене по стойност с foreach

```
Console.WriteLine("\nVavedi uspeh na student");  
inp = Convert.ToInt32(Console.ReadLine());  
foreach (DictionaryEntry st in sl)  
{  
    if ((int)st.Value == inp)  
        Console.WriteLine("Student {0}, uspeh {1}", st.Key, st.Value);  
}  
Console.ReadKey();  
}
```



# Kлас SortedList

```
file:///D:/Violeta_new/C#proekti1/ConsoleA...
Student Ivan, uspeh 6
1. *****
Student Dragan, uspeh 2
Student Ina, uspeh 5
Student Ivan, uspeh 3
Student Kalin, uspeh 6
2. *****
Dragan 2
Ina 5
Ivan 3
Kalin 6

3. U spisaka sa samo slednite imena:
Dragan
Ina
Ivan
Kalin

4. Imena - uspeh
Dragan - 2
Ina - 5
Ivan - 3
Kalin - 6

Uvedi ime na student
Ina
5

Uvedi ime na student
Ina
Student Ina, uspeh 5

Uvedi uspeh na student
3
e u spisaka!!!

Uvedi uspeh na student
3
Student Ivan, uspeh 3
```



# Специални и силно типизирани колекции в `System.Collections.Specialized`

- В `System.Collections.Specialized` се намират някои специални колекции, които приличат на разгледаните по-горе, но имат малко по-специално предназначение.
- Пример за специален тип колекция е хеш-таблица от низове, която не прави разлика между главни от малки букви в ключа на елементите. В .NET Framework такава колекция можем да получим чрез метода `CreateCaseInsensitiveHashtable()` на класа `CollectionsUtil`:



```
using System;
using System.Collections; // за клас Hashtable
using System.Collections.Specialized;
//за клас CollectionsUtil
class Program
{
    static void Main(string[] args)
    {
        Hashtable names =
            CollectionsUtil.CreateCaseInsensitiveHashtable();
        names["Ivan"] = "Ivanov";
        Console.WriteLine("{0} ima familia {1}",
            "Ivan", names["ivan"]);
        Console.ReadLine();
    }
}
```

Резултат: Ivan ima familia Ivanov



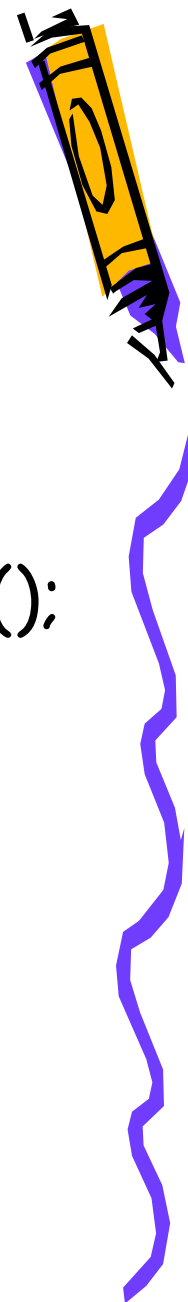
# StringDictionary

- В **System.Collections.Specialized** има и силно типизирани колекции.
- Пример за силно типизирана колекция в **System.Collections.Specialized** е **StringDictionary**.
- Това е клас, който работи точно като **Hashtable**, но използва само **string** за ключове и стойности. При използване на този клас няма нужда от преобразуване на типа към **string** при извличане на стойност от хеш-таблицата:



```
using System;
using System.Collections.Specialized;
// класа StringDictionary
class Program
{
    static void Main(string[] args)
    {
        StringDictionary names = new StringDictionary();
        names["Ivan"] = "Ivanov";
        names["Simona"] = "Peeva";
        Console.WriteLine("{0} ima familia {1}",
            "Ivan", names["Ivan"]);
        Console.ReadLine();
    }
}
```

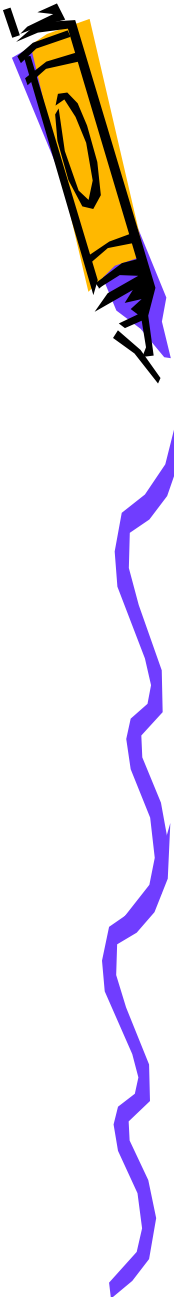
Резултат: Ivan ima familia Ivanov





# StringCollection

- Друга силно типизирана колекция в `System.Collections.Specialized` е `StringCollection` - аналог на `ArrayList`, но за `string` обекти.



```
using System;
using System.Collections.Specialized; // класа
StringCollection
class Program
{static void Main(string[] args)
{  StringCollection list = new StringCollection();
  list.Add("Apples");
  list.Add("Oranges");
  list.Add("Kiwi");
  for (int j=0;j<list.Count;j++)
  { Console.Write("{0} ", list[j]); }
  Console.ReadLine();
}
}
```

 Результат: Apples Oranges Kiwi

