

ДЕЛЕГАТИ, СЪБИТИЯ, ЛАМБДА ФУНКЦИИ И ИНТЕГРИРАН ЕЗИК ЗА ЗАЯВКИ LINQ

ЛЕКЦИЯ 3

Съдържание

- ▶ Делегати (delegates). Дефиниране, инстанциране, извикване
- ▶ Single-cast и multicast делегати
- ▶ Събития (events)
- ▶ Разлика между събитие и инстанция на делегат
- ▶ Разширяващи методи (extension methods)
- ▶ Анонимни типове (anonymous types)
- ▶ Ламбда изрази
- ▶ Колекции
- ▶ Интегриран език за заявки LINQ

Какво представляват делегатите?

- ▶ Делегатите представляват **.NET** типове, които описват:
 - ▶ **сигнатурата на даден метод** (брой, тип и последователност на параметрите му) и
 - ▶ **връщания от него тип**
- ▶ Делегатите съдържат силно типизиран указател (**референция**) към метод
 - ▶ **както** указателите към функции в C++
- ▶ Те са **структури от данни**, които **приемат** като стойност **методи**, отговарящи на описаната от делегата сигнатура
- ▶ Чрез тях се осъществяват "**обратни извиквания**" (callbacks)
- ▶ Могат да сочат **както** към **статични методи**, така и към **методи на инстанция**

Пример за делегат – задача Lect3_zad1

```
// Declaration of a delegate
public delegate void SimpleDelegate(string aParam);

class TestDelegate
{
    public static void TestFunction(string aParam)
    {
        Console.WriteLine("I was called by a delegate.");
        Console.WriteLine("I got parameter {0}.", aParam);
    }
    public static void Main()
    {
        // Instantiation of a delegate
        SimpleDelegate simpleDelegate = new SimpleDelegate(TestFunction);

        // Извикване на метода, посочен от делегата
        simpleDelegate("test");

        Console.ReadKey();
    } //end Main()
}
```

Видове делегати

- ▶ Делегатите в **.NET Framework** са **специални класове** и могат да бъдат два вида:
 - ▶ **Единични (single-cast)** делегати
 - ▶ Съдържат **референция** към един **единствен метод**
 - ▶ Наследяват класа **System.Delegate**
 - ▶ **Множествени (multicast)** делегати
 - ▶ Съдържат **свързан списък** от **референции** към **методи**
 - ▶ Наследяват класа **System.MulticastDelegate**
- ▶ В C# могат да се декларират **само Multicast** делегати чрез запазената дума **delegate**

Делегати – пример

```
namespace Lect3_zad1
{
    public delegate int BinaryOp(int x, int y);
    class SimpleMath
    {
        public static int Add(int x, int y)
        { return x + y; }
        public static int Subtract(int x, int y)
        { return x - y; }
    }
    static void Main(string[] args)
    {
        Console.WriteLine("***** Simple Delegate Example *****\n");
        // Create a BinaryOp delegate object that "points to"
        // SimpleMath.Add()
        BinaryOp b = new BinaryOp(SimpleMath.Add);
        // Invoke Add() method indirectly using delegate object
        Console.WriteLine("10 + 10 is {0}", b(10, 10));
        Console.ReadLine();
    }
}
```

Multicast делегати

- ▶ При извикване на **multicast** делегат, се изпълняват **последователно** един след друг **всички методи** от неговия списък
- ▶ Ако **multicast** делегат **връща стойност** или **променя ref** или **out** параметър, резултатът е само от **последния извикан метод** от списъка с методи на делегата
- ▶ Ако при извикване на **multicast** делегат някой от методите в неговия списък **хвърли изключение**, **следващите методи** от списъка **не се извикват**
- ▶ На практика **single-cast** делегати **почти не се използват** и под **делегат** обикновено се има предвид **multicast делегат**

System.MulticastDelegate

- ▶ Класът `System.MulticastDelegate`
 - ▶ е наследник на `System.Delegate` и е **базов клас** за всички делегати в C#
 - ▶ съдържа **метод `Combine`** за **сливане на списъците** от методи на няколко делегата от еднакъв тип
 - ▶ съдържа **метод `Remove`** за **премахване на метод** от списъка за извикване
 - ▶ има **метод `GetInvocationList()`**, който връща **масив от делегати** – по един за всеки от методите в списъка за извикване на делегата
 - ▶ има **свойство `Method`**, което **описва сигнатурата на методите** в делегата

Multicast делегати – пример

```
public delegate void StringDelegate(string aValue);

public class TestDelegateClass
{
    void PrintString(string aValue)
    {
        Console.WriteLine(aValue);
    }

    void PrintStringLength(string aValue)
    {
        Console.WriteLine("Length = {0}", aValue.Length);
    }

    static void PrintStringWithDate(string aValue)
    {
        Console.WriteLine("{0}: {1}", DateTime.Now, aValue);
    }
}
```

(примерът продължава)

Multicast делегати – пример

```
static void PrintInvocationList(Delegate aDelegate)
{
    Console.Write("(");
    Delegate[] list = aDelegate.GetInvocationList();
    foreach (Delegate d in list)
        Console.Write(" {0}", d.Method.Name);
    Console.WriteLine(")");
}

public static void Main(String[] args)
{
    TestDelegateClass tdc = new TestDelegateClass();

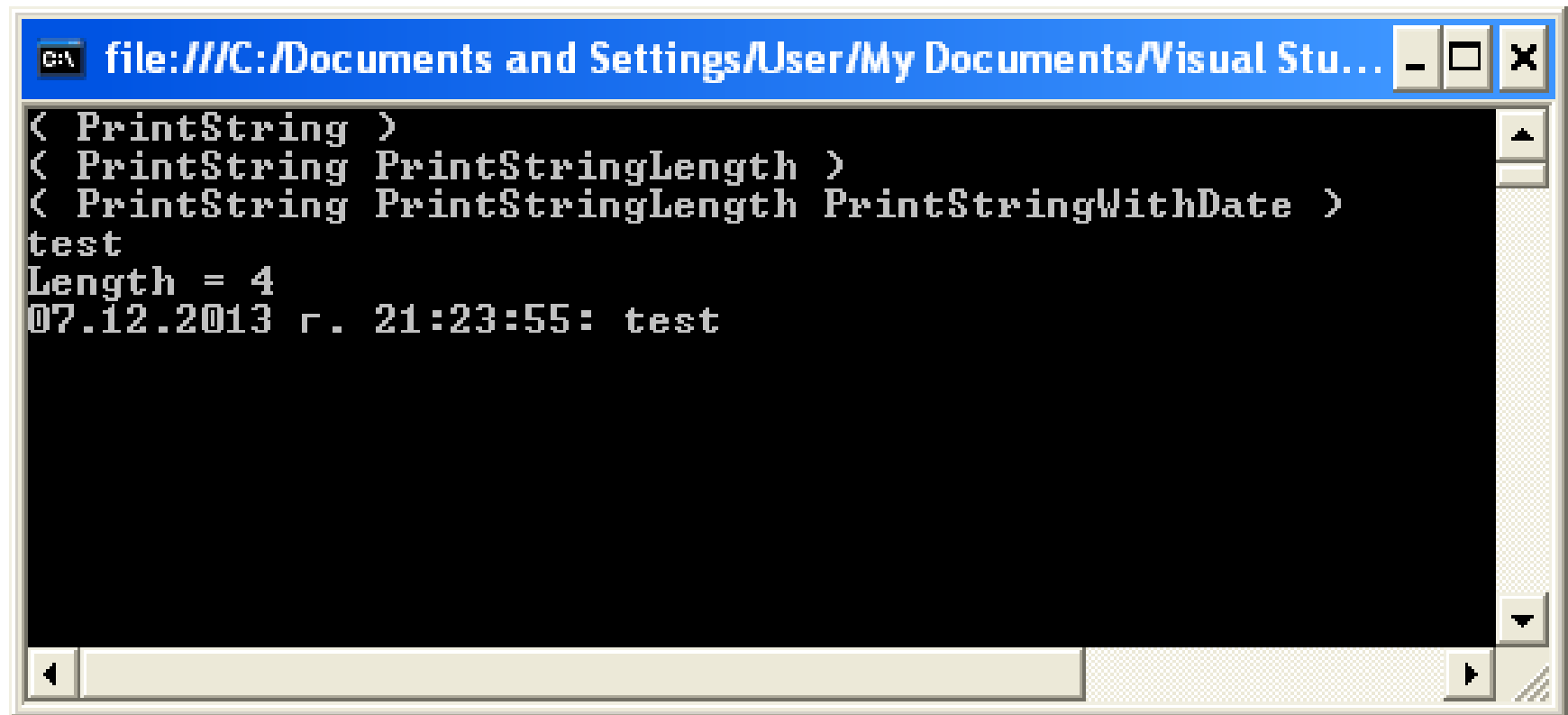
    StringDelegate printDelegate =
        new StringDelegate(tdc.PrintString);
    StringDelegate printLengthDelegate =
        new StringDelegate(tdc.PrintStringLength);
    StringDelegate printWithDateDelegate =
        new StringDelegate(PrintStringWithDate);
}
```

(примерът продължава)

Multicast делегати – пример

```
PrintInvocationList(printDelegate);  
// Prints: ( PrintString )  
  
StringDelegate combinedDelegate = (StringDelegate)  
    Delegate.Combine(printDelegate,  
        printLengthDelegate);  
  
PrintInvocationList(combinedDelegate);  
// Prints: ( PrintString PrintStringLength )  
  
combinedDelegate = (StringDelegate)  
    Delegate.Combine(combinedDelegate,  
        printWithDateDelegate);  
  
PrintInvocationList(combinedDelegate);  
// Prints: ( PrintString PrintStringLength  
//           PrintStringWithDate )  
  
// Invoke the delegate  
combinedDelegate("test");  
}  
}
```

Резултати:



A screenshot of a Visual Studio console window. The title bar is blue and contains the text "file:///C:/Documents and Settings/User/My Documents/Visual Stu..." followed by standard window control buttons (minimize, maximize, close). The console area has a black background with white text. The output shows three lines of test cases, each starting with a less-than sign and followed by the function name in angle brackets. The first two lines are for "PrintString" and "PrintStringLength". The third line is for "PrintStringWithDate". Below these, the word "test" is printed, followed by "Length = 4" and a timestamp "07.12.2013 г. 21:23:55:" followed by "test".

```
< PrintString >  
< PrintString PrintStringLength >  
< PrintString PrintStringLength PrintStringWithDate >  
test  
Length = 4  
07.12.2013 г. 21:23:55: test
```

Събития (events)

- ▶ В компонентно-ориентираното програмиране компонентите изпращат **събития (events)** към своя притежател, за да го **уведомят при настъпване** на интересна за него **ситуация**
- ▶ **Обектът**, който **предизвиква дадено събитие**, се нарича **изпращач на събития** (event sender)
- ▶ **Обектът**, който **получава дадено събитие**, се нарича **получател на събитието** (event receiver)
- ▶ За да **получават** дадено събитие получателите му предварително се **абонира**т за него (subscribe for event)

Събития в .NET

- ▶ В компонентния модел на .NET Framework **абонирането, изпращането и получаването** на събитията се поддържа чрез **делегати и събития**
- ▶ **Събитията** в C# са **специални инстанции на делегати**, декларирани с **ключовата дума event**
- ▶ За **променливите от тип събитие**, C# компилаторът **автоматично дефинира** операторите **+=** и **-=**, съответно **за абониране** за събитието и за **премахване на абонамент**
- ▶ **Събитията** могат да **предефинират кода** за абониране и премахване на абонамент

Разлика между събитие и делегат

- ▶ **Събитията**, декларирани с ключовата дума **event** **не са еквивалентни** на **член-променливите** от тип **делегат**

```
public MyDelegate m;
```

≠

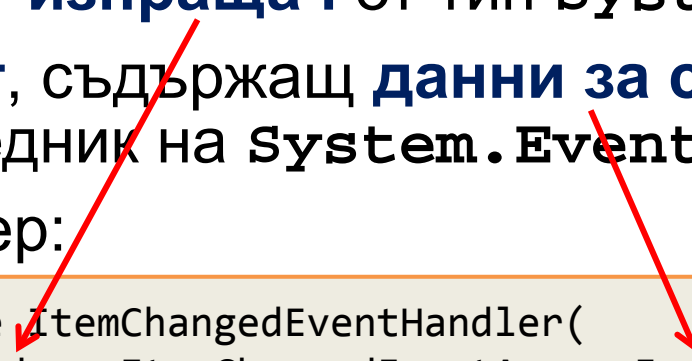
```
public event MyDelegate m;
```

- ▶ Събитията **могат** да бъдат **членове на интерфейс**, а делегатите **не могат**
- ▶ Извикването на **събитие** може да става **само** от **класа**, в който е **дефинирано**
- ▶ **Достъпът до събитията** по подразбиране е **синхронизиран**

Конвенция за събитията

- ▶ В .NET Framework се използва **утвърдена конвенция** за събитията:
- ▶ **Делегатите**, които се използват за **събития**:
 - ▶ имат **имена** образувани от глагол + `EventHandler` (*`SomeVerbEventHandler`*)
 - ▶ връщат `void` и приемат **два параметъра**:
 - **обект-изпращач** от тип `System.Object` и
 - **обект**, съдържащ **данни за събитието** от тип, наследник на `System.EventArgs`
 - ▶ пример:

```
public delegate ItemChangedEventHandler(  
    object aSender, ItemChangedEventArgs aEventArgs);
```



Конвенция за събитията

- ▶ Събитията се обявяват като **public**, започват с **главна буква** и завършват с **глагол**, например:

```
public event ItemChangedEventHandler ItemChanged;
```

- ▶ За предизвикване на събитие се създава **protected void** метод с име в стил **OnVerb**, например:

```
protected void OnItemChanged() { ... }
```

- ▶ Методът-получател (**обработчик**) на събитието има име **Обект_Събитие**:

```
private void OrderList_ItemChanged() { ... }
```

Събития – примен

определя типа
на EventHandler

```
using System;
public delegate void EventHandler();
class Program { public static event EventHandler Print;
static void Main()
{
    // Add event handlers to Show event.
    Print += new EventHandler(Dog);
    Print += new EventHandler(Cat);
    Print += new EventHandler(Mouse);
    Print += new EventHandler(Mouse);
    // Invoke the event.
    Print.Invoke();
}
static void Cat()
{
    Console.WriteLine("Cat");
}
static void Dog()
{
    Console.WriteLine("Dog");
}
static void Mouse()
{
    Console.WriteLine("Mouse");
}
}
```

**+= абониране
за събитието**

Изход

Cat

Dog

Mouse

Mouse

Събития – пример

- **Метод-получател (обработчик)** на събитието

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(textBox1.Text);
}
private void button2_Click(object sender, EventArgs e)
{
    textBox1.Text = DateTime.Now.ToString();
}
```

- **Събитията** ви позволяват да **добавите методи**, които се задействат при външно събитие
- Може да добавяте **нови обработчици на събития динамично за конкретни действия**

Делегатът `System.EventHandler`

▶ Делегатът `System.EventHandler`

```
public delegate void EventHandler( Object sender, EventArgs e);
```

- ▶ дефинира **референция към callback метод**, който **обработва събития**, за които не се изпраща допълнителна информация
- ▶ на много места се **използва** вътрешно от .NET Framework

▶ Класът `EventArgs`

- ▶ **не съдържа** никаква информация за събитието – **той е базов клас**.
- ▶ **неговите наследници** съдържат информация за **настъпили събития**

System.EventHandler – пример

```
public class Button
{
    public event EventHandler Click;
    public event EventHandler GotFocus;
    public event EventHandler TextChanged;
    ...
}

public class ButtonTest
{
    private static void Button_Click(object aSender,
        EventArgs aEventArgs)
    {
        Console.WriteLine("Button_Click() event called.");
    }

    public static void Main()
    {
        Button button = new Button();
        button.Click += new EventHandler(Button_Click);
        button.DoClick();
    }
}
```

Събития и интерфейси

- ▶ Събитията (events) могат да бъдат **членове на интерфейси**:

```
public interface IClickable
{
    event ClickEventHandler Click;
}
```

- ▶ При **имплементацията на събитие** от интерфейс за него могат да се реализират специфични **add** и **remove** методи
- ▶ За разлика от свойствата, при събитията **имплементацията на методите add и remove не е задължителна**

Разширяващи методи (**extension methods**)

- ▶ Често пъти в практиката се налага да се **добавя нова функционалност** към съществуващ код
- ▶ Ако кодът е наш, добавяме нужната функционалност и да **прекомпилираме**
- ▶ Когато дадено **асембли** (.exe или .dll файл) е вече компилирано, и кодът не е наш, класическият вариант за разширяване на функционалността на типовете е чрез **наследяване**
- ▶ Ако типът, който искаме да наследим е маркиран с ключовата дума **sealed**, то **опция за наследяване няма**
- ▶ **Разширяващите методи (extension methods)** решават точно този проблем
 - ▶ дават възможност да **добавяме функционалност към съществуващ тип** (клас или интерфейс),
 - ▶ без да променяме оригиналния му код и дори без наследяване

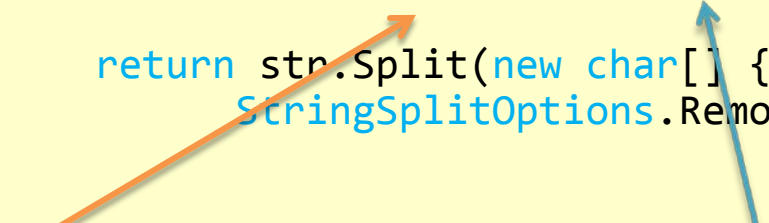
Разширяващи методи

- ▶ **Разширяващите методи** се дефинират като **статични методи** в обикновени статични класове
- ▶ Типът на първия им аргумент представлява **класа** (или интерфейса), **към който се закачат**
- ▶ Преди него се поставя ключовата дума **this**
- ▶ Това ги **отличава** от другите статични методи и показва на компилатора, че това е **разширяващ метод**
- ▶ **Параметърът**, пред който **стои ключовата дума this**, може да бъде **използван в тялото на метода**, за да се **създаде функционалността** на метода
- ▶ Той реално представлява **обекта**, с който разширяващият метод работи

Разширяващи методи – примери

- ❖ За да могат да бъдат достъпни дадени разширяващи методи, трябва да бъде добавен с **using** съответният **namespace**, в който е дефиниран статичния клас, описващ тези методи
- ▶ В противен случай, компилаторът няма как да разбере за тяхното съществуване

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(new char[] { ' ', '.', '?', '!' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```



- ▶ **Разширяващ метод**, който брои колко думи има в даден текст (**string**)
 - ▶ Методът **WordCount()** се закача за класа **string**
- ▶ Това е оказано с ключовата дума **this** преди типа и името на първия аргумент на метода (в случая **str**)
- ▶ Самият метод е статичен и е дефиниран в статичния

Разширяващи методи – примери

```
static void Main()
{
    string helloString = "Hello, Extension Methods!";
    int wordCount = helloString.WordCount();
    Console.WriteLine(wordCount);
}
```

- ▶ Самият метод се вика върху **обекта** **helloString**, който е от тип **string**
- ▶ Методът получава **обекта като аргумент** и работи с него
- ▶ В случая вика неговия метод **Split(...)** и връща **брой елементи в получения низ (масив)**

Анонимни типове (anonymous types)

- ▶ В обектно-ориентирани езици (каквото е C#) много често се налага да се дефинират малки класове с цел еднократно използване
- ▶ Типичен пример за това е класът **Point**, съдържащ само 2 полета – координатите на точка
- ▶ В езика C# има вграден начин за създаване на типове за еднократна употреба, наричани **анонимни типове (anonymous types)**
- ▶ С ключовата дума **var** показваме на компилатора, че типа на променливата трябва да се разбере автоматично от дясната страна на присвояването

Анонимни типове – пример

```
var myCar = new { Color = "Red", Brand = "BMW", Speed = 180 };
```

- ▶ За свойствата **Color** и **Brand** компилаторът сам ще се досети, че са от тип **string**, а за свойството **Speed**, че е от тип **int**.
- ▶ Веднага след инициализацията си, обектът от **АНОНИМНИЯ ТИП** може да бъде **ИЗПОЛЗВАН КАТО ОБИКНОВЕН ТИП** с трите си свойства:

```
Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Brand);  
Console.WriteLine("It runs {0} km/h.", myCar.Speed);
```

- ▶ Резултатът от изпълнението ще е следния:

```
My car is a Red BMW.  
It runs 180 km/h.
```

ЛАМБДА ИЗРАЗИ (lambda expressions)

- ▶ **Ламбда изразите** представляват **анонимни функции**, които съдържат **изрази** или **последователност от оператори**
- ▶ **Всички ламбда изрази** използват **ламбда оператора =>**, който може да се чете като "**отива в**"
 - ▶ Идеята за ламбда изразите в C# е взaimствана от функционалните езици (например **Haskell**, **Lisp**, **Scheme**, **F#** и др.)
- ▶ Обикновено **ламбда изразите** се използват като предикати или **вместо делегати** (променливи от тип функция), които се **прилагат върху колекции**, обработвайки елементите от колекцията по някакъв начин и/или връщайки определен резултат

ЛАМБДА ИЗРАЗИ

- ▶ Лявата страна на **лямбда оператора** определя **входните параметри на анонимната функция**
- ▶ Дясната страна представлява **израз** или **последователност от оператори**, която работи с входните параметри и евентуално връща някакъв резултат

```
// [Аргументи] => [тяло на метода]
```

```
// С въведени параметри
```

```
n => n == 2
```

```
(a, b) => a + b
```

```
(a, b) => { a++; return a + b; }
```

```
// С изрично посочени параметри
```

```
(int a, int b) => a + b
```

```
// Без въведени параметри
```

```
() => return 0
```

```
// Присвояване на лямбда израз на делегат
```

```
Func<int, int, int> f = (a, b) => a + b;
```

ЛАМБДА ИЗРАЗИ

- ▶ Телата на **ламбда изрази**, съдържащи **повече от един израз**, са **оградени от скоби**
- ▶ Вътре в скобите, кодът може да бъде написан като стандартен метод

```
(a, b) => { a++; return a + b; }
```

- ▶ **Ламбда изразите** могат да бъдат подавани и като **аргументи при извикването на метод**, подобно на анонимни делегати

```
var list = stringList.Where(n => n.Length > 2);
```

Ламбда изрази – примери

- ▶ Нека да разгледаме разширяващия метод **FindAll(...)**, който може да се използва за филтриране на необходимите елементи
 - ▶ Той работи върху определена **колекция**, прилагайки ѝ даден предикат, който проверява **всеки от елементите** на колекцията **дали отговаря на определено условие**
 - ▶ За да го използваме, трябва да включим **референция към библиотеката System.Core.dll** и **namespace-a System.Linq**
- ▶ Ако искаме да вземем **само четните числа** от колекция с цели числа, можем да използваме метода **FindAll(...)** върху колекцията, като му подадем ламбда метод, който да провери дали дадено число е четно

```
List<int> list = new List<int> () { 1, 2, 3, 4, 5, 6 };  
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);  
foreach (var num in evenNumbers)  
{  
    Console.WriteLine("{0} ", num);  
}  
Console.WriteLine();
```

резултат

2, 4, 6

Колекции

- ▶ **Колекции** наричаме **класовете**, които съдържат **съвкупност от елементи** (т. нар. **контейнер класове**)
- ▶ В .NET Framework класовете, имплементиращи колекции се намират в пространството **System.Collections**
- ▶ **Колекциите в C#** са няколко вида:
 - ▶ **списъчни** (**ICollection**) – ArrayList, Queue, Stack, BitArray, StringCollection
 - ▶ **речникови** (**IDictionary**) – Hashtable, SortedList, StringDictionary
- ▶ За разлика от масивите повечето **колекции имат променлив размер** и
- ▶ позволяват **добавяне и изтриване на елементи**

Колекции (Collections)

- ▶ **.NET Framework Class Library** осигурява няколко класа, наречени **колекции**, които се използват за **запомняне на групи от свързани** обекти
- ▶ Тези класове притежават ефективни методи за **организация, запомняне и обработване на данните**, без да е необходимо да знаете как са съхранени
 - ▶ Това значително **намалява времето за създаване** на приложението
- ▶ Използваме **масиви за запомняне на поредица** от елементи
- ▶ Масивите **не променят автоматично размера си** по време на изпълнение, ако е необходимо да се добави елемент -
 - ▶ ръчно трябва да създадете **нов масив** или да използвате **Array class's Resize method**

Колекции

- ▶ Колекцията **class List<T>** (namespace **System.Collections.Generic**) решава този проблем
- ▶ **Т е контейнер**, когато се декларира нов списък, го **заменя** с **типа на елементите**, които искате да съдържа списъка
- ▶ Това е подобно на определяне на типа при деклариране на масиви
- ▶ Пример:

```
List< int > list1;    //декларира list1 като List collection, която
                    //може да съхранява само цели стойности
List< string > list2 //list2 съхранява само нисове
```

Колекции

- ▶ **Li st<T>** се нарича **generic class**, защото може да се използва с всички типове обекти
- ▶ **T - контейнер** за типа на обектите, съхранени в списъка
- ▶ На следващият слайд са показани **основни методи и свойства** на **class Li st<T>**

Някои методи и свойства на **class List<T>**


Method or property	Description
Add	Adds an element to the end of the List.
Capacity	Property that gets or sets the number of elements a List can store without resizing.
Clear	Removes all the elements from the List.
Contains	Returns true if the List contains the specified element; otherwise, returns false.
Count	Property that returns the number of elements stored in the List.
IndexOf	Returns the index of the first occurrence of the specified value in the List.
Insert	Inserts an element at the specified index.
Remove	Removes the first occurrence of the specified value.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes a specified number of elements starting at a specified index.
Sort	Sorts the List.
TrimExcess	Sets the Capacity of the List to the number of elements the List currently contains (Count).



Колекции – демонстрация

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleAppExtMethods
{
    public class Collection_List
    {
        static void Main(string[] args)
        {
            //create new list of string
            List<string> items = new List<string>();
            items.Add("red");    //добавя елемент към края на List
            items.Insert(0, "yellow"); //вмъква стойност от индекс 0
            items.Insert(1, "green");
            // извеждане на елементите
            Console.WriteLine("Извеждане на колекцията!");
            for (int i = 0; i < items.Count; i++)
                Console.WriteLine("{0}", items[i]);
        }
    }
}
```



Колекции - демонстрация

2 от 3

```
// извеждане на елементи с foreach
Console.WriteLine(" Извеждане на списък с елементи чрез foreach");
foreach (var item in items)
    Console.WriteLine(" {0}", item);
items.Add("white");
items.Add("yellow");
// извеждане на списъка
Console.WriteLine("\n Извеждане с два нови елемента!");
foreach (var item in items)
    Console.WriteLine(" {0}", item);
items.Remove("yellow");
Console.WriteLine("\n Изтрива първият елемент yellow!");
foreach (var item in items)
    Console.WriteLine(" {0}", item);
items.RemoveAt(1);
Console.WriteLine("\n Изтрива елемент с индекс 1  !");
foreach (var item in items)
    Console.WriteLine(" е {0}", item);
```



Колекции - демонстрация

```
foreach (var item in items)
    Console.WriteLine(" {0}", item);
// търсене на стойност в списъка
Console.WriteLine("\n\"green\" is {0} in the list",
    items.Contains("green") ? string.Empty : "not");
//извеждане на брой елементи в списъка
    Console.WriteLine("Брой елементи: {0}", items.Count);
Console.WriteLine("Капацитет:{0}", items.Capacity);
Console.ReadKey();
}
}
```

```
Извеждане на колекцията?
yellow
green
red
Извеждане на списък с елементи чрез foreach
yellow
green
red

Извеждане с два нови елемента?
yellow
green
red
white
yellow

Изтрива първият елемент yellow?
green
red
white
yellow

Изтрива елемент с индекс 1 ?
green
white
yellow

"green" is  in the list
Брой елементи: 3
Капацитет:8
```


ArrayList – пример

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
namespace ConsoleApprrayList
{
    public class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            for (int i = 1; i <= 10; i++)
            {
                list.Add(i);           // Добавяне i в края
            }
            list.Insert(3, 123);        // Вмъкваме 123 преди елемент 3
            list.RemoveAt(7);           // Премахваме елемент с индекс 7
            list.Remove(2);             // Премахваме елемент със стойност 2
            list[1] = 500;              // Променяме елемент с индекс 1
            list.Sort();               // Сортираме в нарастващ ред
            int[] arr = (int[])list.ToArray(typeof(int));
            foreach (int i in arr)
            {
                Console.Write("{0} ", i);
            }
            Console.WriteLine();
            Console.ReadKey();
            // Резултат: 1 4 5 6 8 9 10 123 500
        }
    }
}
```

Други списъчни колекции

- ▶ **Queue** – опашка (**first-in, first-out структура**), реализирана с цикличен масив, поддържа:
 - ▶ добавяне на елемент (Enqueue)
 - ▶ извличане на елемент (Dequeue)
- ▶ **Stack** – стек (**last-in, first-out структура**), реализиран с масив, поддържа:
 - ▶ добавяне на елемент (Push)
 - ▶ извличане на елемент (Pop)
 - ▶ преглеждане на елемент (Peek)
- ▶ **StringCollection** – като ArrayList, но за string обекти
- ▶ **BitArray** – масив от булеви стойности, всяка записана в 1 бит

LINQ заявки (LINQ queries)

- ▶ **LINQ (Language-Integrated Query)** представлява редица **разширения** на .NET Framework, които включват **интегрирани в езика заявки и операции върху елементи** от даден източник на данни (най-често **масиви и колекции**)
- ▶ **LINQ** е много мощен инструмент, който доста прилича на повечето SQL езици и по синтаксис и по логика на изпълнение
- ▶ За да се използват **LINQ заявки** в езика C#, трябва да включим референция към **System.Core.dll** и да добавим namespace-а **System.Linq**

LINQ

- ▶ **LINQ** предоставя възможности да се пишат заявки - изрази, които **извличат информация** от **различни източници на данни**, а не само от бази данни
- ▶ **LINQ** към обекти може да се използва за **филтриране** на **масиви (arrays)** и **Lists (колекции)** - избор на елементи, които да **отговарят на набор от условия**
- ▶ **LINQ** предоставя **набор от класове**, които реализират **LINQ операции** и дават възможност програмата да си взаимодейства с **източници на данни**, за да изпълнява задачи като **проектиране, сортиране, групиране и филтриране на елементи**

Language-Integrated Query (LINQ)

- ▶ Набор от специални средства, въведени във Visual Studio 2008, които разширяват възможността за заявки в синтаксиса на C# и Visual Basic
- ▶ **LINQ въвежда стандарт**, лесен за изучаване шаблон (**patterns**) за изпълнение на заявки и обновяване на данни, както и технология, която може да се разширява за поддържане на различни видове данни
- ▶ **Visual Studio включва LINQ доставчик**, който осигурява използването на LINQ към
 - ▶ **.NET Framework collections**
 - ▶ **SQL Server databases**
 - ▶ **ADO.NET Datasets**
 - ▶ **XML documents**



Избор на източник на данни с LINQ

Пример: ConsoleAppLinq

- ▶ С ключовите думи **from** и **in** се задават **източникът на данни** (колекция, масив и т.н.) и **променливата**, с която ще се итерира (обхожда) по колекцията (обхождане по подобие на **foreach** оператора)

- ▶ Например заявка, която започва така:

```
from dataType rangeVar in dataSource
```

```
static void Main(string[] args)
{
    var list = new List<int> { 2, 7, 1, 3, 9, 4, 5};
    var result = from i in list
                  where i > 2
                  select i;
    foreach (int i in result)
    {
        Console.Write("{0} ", i);
    }
    Console.ReadKey();
}

// Резултат: 7 3 9 4 5
```

Филтриране на данните с LINQ

```
string[] array = { "dot", "", "net", null, null, "perls", null };  
// Use Where method to remove null strings.  
var result1 = array.Where(item => item != null);
```

- ▶ С ключовата дума **where** се задават условията, които всеки от елементите от колекцията трябва да изпълнява, за да продължи да се изпълнява заявката за него
- ▶ Изразът след **where** винаги е **булев израз** (`where i > 2`)
- ▶ Може да се каже, че с **where** се **филтрират елементите**
- ▶ след **where...in** конструкцията използваме само **името**, което сме задали за **обхождане** на всяка една **променлива от колекцията**

```
foreach (string value in result1)  
{  
    Console.WriteLine(value);  
}
```

Избор на резултат от LINQ заявката

- ▶ С ключовата дума **select** се задават **какви данни** да се **върнат от заявката**
- ▶ **Резултатът от заявката** е под формата на **обект** от **съществуващ клас** или **анонимен** тип
- ▶ Върнатият резултат може да бъде и **свойство** на обектите, които заявката обхожда или самите обекти
- ▶ Операторът **select** и всичко след него се поставя винаги в края на заявката
- ▶ Ключовите думи **from**, **in**, **where** и **select** са достатъчни за създаването на проста LINQ заявка

Пример

```
List<int> numbers = new List <int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers =
    from num in numbers
    where num % 2 == 0
    select num;
foreach (var item in evenNumbers)
{
    Console.Write(item + " ");
}
```

- **Заявка** върху **колекция** от **числа** **numbers** и записва в нова колекция **само четните числа**
- Заявката може да се прочете така:
 - ❖ *За всяко число **num** от **numbers** провери дали се дели на 2 без остатък и ако е така го добави в новата колекция*
- **Резултат:** 2 4 6 8 10

Сортиране на данните с LINQ

- ▶ **Сортирането** чрез LINQ заявките се извършва с ключовата дума **orderby**
- ▶ След нея се слагат условията, по които да се подреждат елементите, участващи в заявката
 - ▶ за всяко условие може да се укаже редът на подреждане: в **нарастващ ред** - с ключова дума **ascending**
 - ▶ **намаляващ ред** - с ключова дума **descending**
 - ▶ по **подразбиране** се подреждат в **нарастващ ред**

Пример: сортиране на масив от думи (низове) по дължината им в намаляващ ред

```
string[] words = { "cherry", "apple", "blueberry" };  
var wordsSortedByLength =  
    from word in words  
    orderby word.Length descending  
    select word;  
foreach (var word in wordsSortedByLength)  
{  
    Console.WriteLine(word);  
}
```

► Резултатът е:

```
blueberry  
cherry  
apple
```

- Ако не бъде указан начина, по който да се сортират елементите (т.е. ключовата дума **orderby** отсъства от заявката), се взимат елементите в реда, в който колекцията би ги върнала при обхождане с **foreach** оператора

Групиране на резултатите с LINQ

- ▶ С ключовата дума **group** се извършва групиране на резултатите по даден критерии
- ▶ Форматът е следният:
group [име на променливата] **by** [признак за групиране] **into** [име на групата]
- ▶ Резултатът от това групиране е **нова колекция от специален тип**, която може да бъде използвана по-надолу в заявката
- ▶ След групирането, заявката спира да работи с първоначалната си променлива
- ▶ Това означава, че в **select**-а може да се ползва само в групата

```

int[] numbers =
    { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0, 10, 11, 12, 13 };
int divisor = 5;
var numberGroups =
    from number in numbers
    group number by number % divisor into group
    select new { Remainder = group.Key, Numbers = group };
foreach (var group in numberGroups)
{
    Console.WriteLine(
        "Numbers with a remainder of {0} when divided by {1}:",
        group.Remainder, divisor);
    foreach (var number in group.Numbers)
    {
        Console.WriteLine(number);
    }
}

```

- ▶ На конзолата се извеждат числата, групирани по остатъка си от деление с 5.
- ▶ В заявката за всяко число се смята `number % divisor` и за всеки различен резултат се прави нова група.
- ▶ По-надолу **select** оператора работи върху списъка от създадените групи и за всяка група създава анонимен тип, който съдържа 2 свойства: **Remainder** и **Numbers**.
- ▶ На свойството **Remainder** се присвоява **ключа** на групата (остатъка от деление с **divisor** на числото)
- ▶ На свойството **Numbers** се присвоява **колекцията group**, която съдържа всички елементи в групата.
- ▶ **select**-а се изпълнява само и единствено върху списъка от групи

Резултат

Numbers with a remainder of 0 when divided by 5:

5

0

10

Numbers with a remainder of 4 when divided by 5:

4

9

Numbers with a remainder of 1 when divided by 5:

1

6

11

Numbers with a remainder of 3 when divided by 5:

3

8

13

Numbers with a remainder of 2 when divided by 5:

7

2

12

Пример за използване на LINQ

ConsoleAppLINQ Zad2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleAppLINQ_Zad2
{
    public class Program
    {
        static void Main(string[] args)
        {
            int [] values = {2, 9, 5, 0, 2, 7, 1, 8};
            Console.WriteLine("Оригинален масив! \n");
            foreach (var element in values)
                Console.WriteLine("{0} ", element);
            // LINQ елементи по-големи от 4
            var filtered =
                from value in values
                where value > 4
                select value;
            Console.WriteLine("\n Филтриран масив element > 4 ");
            foreach (var element in filtered)
                Console.WriteLine("{0} ", element);
        }
    }
}
```

Пример за използване на LINQ

```
Console.WriteLine("\n Сортиран масив във възходящ ред !");
    var sortarr =
        from value in values
        orderby value
        select value;
    foreach (var element in sortarr)
        Console.Write("{0} ", element);
var sortandfilter =
    from value in values
    where value > 4
    orderby value descending
    select value;
    // Izwevda Sortiran masiv
Console.WriteLine("\n Сортиран и филтриран масив във низходящ ред !");
foreach (var element in sortandfilter)
    Console.Write("{0} ", element);
Console.ReadKey();
}
```


Пример за използване на LINQ

```
Оригинален масив!  
2 9 5 0 2 7 1 8  
Филтриран масив element > 4  
9 5 7 8  
Сортиран масив във възходящ ред !  
0 1 2 2 5 7 8 9  
Сортиран и филтриран масив във низходящ ред !  
9 8 7 5
```

Съединение на данни с LINQ

- ▶ Операторът **join** има доста по-сложна концепция от останалите LINQ оператори
- ▶ **Съединява колекции по даден критерий** (еднаквост) между тях и извлича необходимата информация от тях
- ▶ Синтаксисът му е следният:
from [име на променлива от колекция 1] **in** [колекция 1]
join [име на променлива от колекция 2] **in** [колекция 2]
on [част на условието за еднаквост от колекция 1]
equals [част на условието за еднаквост от колекция 2]
- ▶ По-надолу в заявката (в **select**-а например) може да се използва, както името на променливата от колекция 1, така и това от колекция 2

Съединение на данни с LINQ – пример

```
class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
}
class Book
{
    public int AuthorId { get; set; }
    public string Title { get; set; }
}
// Вижда се, че двата класа имат едно и също свойство AuthorId
```

- ▶ Вижда се, че двата класа имат едно и също свойство **AuthorId**
- ▶ Това ще бъде свойството, което ще определи съответствието между обект Book и обект Author

```

class Program
{
    public static void Main(string[] args)
    {
        Author[] authors = new Author[]
        {
            new Author() { AuthorId = 1, Name = "John Smith" },
            new Author() { AuthorId = 2, Name = "Harry Gold" },
            new Author() { AuthorId = 3, Name = "Ronald Schwimmer" },
            new Author() { AuthorId = 4, Name = "Jerry Mawler" }
        };
        Book[] books = new Book[]
        {
            new Book() { AuthorId = 1, Title = "Little Blue Riding Hood" },
            new Book() { AuthorId = 3, Title = "The Three Little Piggy Banks" },
            new Book() { AuthorId = 1, Title = "Snow Black" },
            new Book() { AuthorId = 2, Title = "My Rubber Duckie" },
            new Book() { AuthorId = 2, Title = "He Who Doesn't Know His Name" },
            new Book() { AuthorId = 1, Title = "Hanzel and Brittle" }
        };
        var result = from a in authors
                     join b in books on a.AuthorId equals b.AuthorId
                     select new { a.Name, b.Title };
        foreach (var r in result)
        {
            Console.WriteLine("{0} - {1}", r.Name, r.Title);
        }
    }
}

```



Резултат

```
John Smith - Little Blue Riding Hood  
John Smith - Snow Black  
John Smith - Hanzel and Brittle  
Harry Gold - My Rubber Duckie  
Harry Gold - He Who Doesn't Know His Name  
Ronald Schwimmer - The Three Little Piggy Banks
```

- ▶ Клаузата **join**, като всяка друга клауза се превежда по време на компилация в извикване на Join метод

```
var result = authors.Join(books,  
    author => author.AuthorId,  
    book => book.AuthorId,  
    (author, book) => new { author.Name, book.Title }  
);
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!