Objektovo orientované programovanie v C#

ERIK KUČERA

PROGRAMOVANIE GUI | PREDNÁŠKA (TÝŽDEŇ 3)

Statické členy

- Statické členy nie sú takým pilierom OOP ako dedičnosť alebo polymorfizmus, ale sú užitočným prvkom
- K statickým členom a metódam môžete pristupovať aj keď nemáte vytvorenú žiadnu inštanciu, dokonca môžete vytvárať celé statické triedy

```
class Pes {
        private String meno; // ked dame psovi meno, uz ho nenaucime ine
        private static int pocet = 0;

        public Pes(String meno) {
                this.meno = meno;
                pocet++;
        }
        public String getMeno() {
                return meno;
        }
        public static int getPocet() {
                return pocet;
        }
}

Pes dunco = new Pes("Dunco"), e3 = new Pes("Ezechiel Treti");
Pes.getPocet(); // funguje toto
dunco.getPocet(); // aj toto.. ale dava to ovela mensi zmysel
```

Namespace

- Triedy sú v drvivej väčšine organizované do menných priestorov (**namespace**)

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
        }
    }
}
```

Implementácia getterov a setterov

```
double result;

public double Result
{
    get { return result; }
    set { result = value; }
}
```

=

```
public double Result { get; set; }
```

Override a virtual

- „Overridnuť“ môžeme v C# len metódu, ktorá je označená ako **virtual**
- V Jave sú všetky metódy automaticky virtuálne
- V ProgrammerCalculator by sme sa k pôvodnej metóde dostali pomocou kľúčového slova **base** – napríklad **base.Add(num1, num2)**

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}

public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);
```

Abstraktné triedy

- Nemôžeme vytvoriť objekt **abstraktnej triedy**, musíme tak vytvoriť triedu, ktorá ju bude dediť
- Ide o niečo podobné ako interface (bude vysvetlené ďalej)
- Abstraktná trieda môže obsahovať tiež **abstraktné metódy**, ktoré treba implementovať v odvodených triedach, tieto metódy sú tak implicitne virtuálne

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

Interface

- **Interface** je vlastne abstraktná trieda, avšak nemá implementované žiadne metódy
- Implementáciu treba napísať do odvodenej triedy
- Trieda môže implementovať viacero interfacov (abstraktnú resp. hocijakú triedu len jednu)
- Zvyknú sa pomenúvať tak, že prvé písmeno je `I`

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Partial class

Two classes can be in the same namespace

Take a look at these two class files from a program called `PetFiler2`. They've got three classes: a `Dog` class, a `Cat` class, and a `Fish` class. Since they're all in the same `PetFiler2` namespace, statements in the `Dog.Bark()` method can call `Cat.Meow()` and `Fish.Swim()`. It doesn't matter how the various namespaces and classes are divided up between files. They still act the same when they're run.

SomeClasses.cs

```
namespace PetFiler2 {

  class Dog {

    public void Bark() {
      // statements go here
    }

  }

  partial class Cat {

    public void Meow() {
      // more statements
    }

  }
}
```

When a method is "public" it means every other class in the namespace can access its methods.

MoreClasses.cs

```
namespace PetFiler2 {

  class Fish {

    public void Swim() {
      // statements
    }

  }

  partial class Cat {

    public void Purr() {
      // statements
    }

  }
}
```

Since these classes are in the same namespace, they can all "see" each other—even though they're in different files. A class can span multiple files too, but you need to use the "partial" keyword when you declare it.

You can only split a class up into different files if you use the "partial" keyword. You probably won't do that in any of the code you write in this book, but the IDE used it to split your page up into two files so it could put the XAML code into MainPage.xaml and the C# code into MainPage.xaml.cs.

There's more to namespaces and class declarations, but you won't need them for the work you're doing right now. Flip to #3 in the "Leftovers" appendix to read more.

Generics

ERIK KUČERA

PROGRAMOVANIE GUI | PREDNÁŠKA (TÝŽDEŇ 3)

Aký problém vieme riešiť s generickými typmi?

- Predstavme si, že chceme **implementovať zoznam čísel typu integer** – nižšie vidíme napríklad metódu Add
- Ak chceme implementovať podobný **zoznam kníh**, potrebujeme vytvoriť ďalšiu triedu => **neefektívne**

```
using System;

namespace Generics
{
    public class List
    {
        public void Add(int number)
        {
            throw new NotImplementedException();
        }

        public int this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

```
using System;

namespace Generics
{
    public class BookList
    {
        public void Add(Book book)
        {
            throw new NotImplementedException();
        }

        public Book this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

Neefektívne riešenie problému

- Ako možné riešenie problému sa ponúka nižšie uvedený variant
- Je však z hľadiska výkonu aplikácie veľmi neefektívny

```
public class ObjectList
{
    public void Add(object value)
    {

    }

    public object this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

Riešenie problému pomocou generického typu

- Namiesto konkrétneho typu premennej/objektu napíšeme zátvorky <> a do nich napríklad písmeno `T` ako **template – šablóna**
- Teraz môžeme vytvoriť inštanciu zoznamu pre nami požadovaný typ premennej alebo objektu

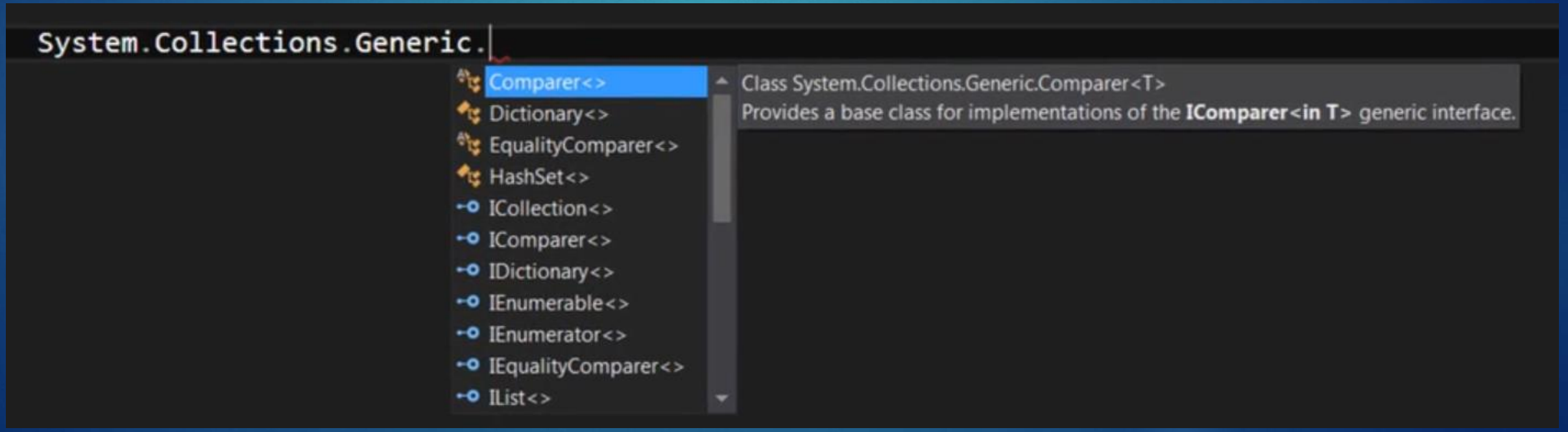
```
public class GenericList<T>
{
    public void Add(T value)
    {

    }

    public T this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var book = new Book { Isbn = "1111", Title = "C# Advanced" };

        //var numbers = new List();
        //numbers.Add(10);

        //var books = new BookList();
        //books.Add(book);

        var numbers = new GenericList<int>();
        numbers.Add(10);

        var books = new GenericList<Book>();
        books.Add(new Book());
    }
}
```

class Generics.Book

Použitie generického typu v praxi

- V praxi budete skôr používať preddefinované generické zoznamy ako vytvárať nové
- Nájdete ich na stránke: https://msdn.microsoft.com/en-us/library/system.collections.generic(v=vs.110).aspx



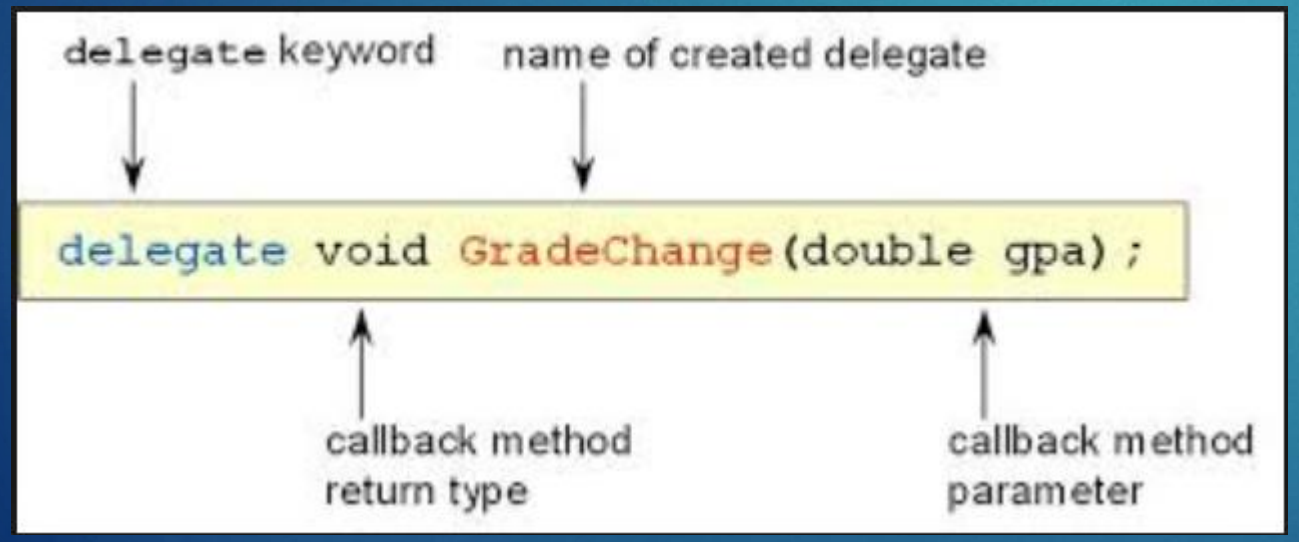
Delegáty, anonymné metódy, lambda výrazy

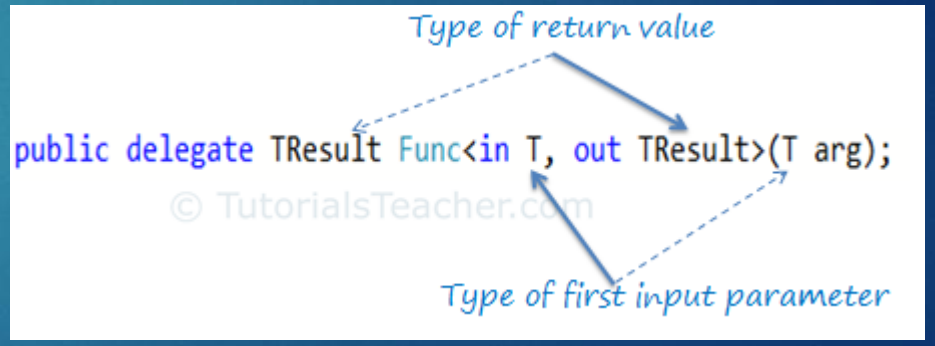
ERIK KUČERA

PROGRAMOVANIE GUI | PREDNÁŠKA (TÝŽDEŇ 3)

Delegát

- Objekt, ktorý vie, ako zavolať metódu alebo skupinu metód
- Dá sa povedať, že ide o referenciu na funkciu
- Užitočný pre dizajn frameworkov a podobne





Pochopenie princípu delegátov

- Predstavme si, že máme metódu, v ktorej chceme urobiť nejakú nešpecifikovanú funkciu, ktorá nám niečo vráti, ale poznáme nejaký jej všeobecný princíp (predpis)
- Keby sme chceli delegáta pre matematické operácie, tak napíšeme:

```
delegate int MathOperation(int a, int b);
```

- Je to predpis pre metódu, ktorá dostane 2 parametre typu `int` a vráti výsledok typu `int`
- Napíšeme si takúto metódu:

```
static void metoda(MathOperation op)
{
    Console.WriteLine(op(1, 2));
}
```

- Pokiaľ chceme zavolať túto metódu, tak jej musíme odovzdať metódu, ktorá spĺňa daný predpis, teda má 2 parametre `int` a vracia `int`:

```
static int Plus (int a, int b)
{
    return a + b;
}
```

- Následne použijeme volanie:

```
static void Main(string[] args)
{
    metoda(Plus);
    Console.ReadKey();
}
```

- A dostaneme výsledok 3

Pochopenie princípu delegátov

- Predstavme si zložitejšiu metódu:

```
static int GetNumber(int[] ints, MathOperation operation)
{
    int result = ints[0];

    for (int i = 1; i < ints.Length; i++)
    {
        result = operation(result, ints[i]);
    }

    return result;
}
```

- Ako parameter dostáva pole integerov a operáciu typu `MathOperation`
- Volanie metódy:

```
static void Main(string[] args)
{
    //metoda(Plus);
    Console.WriteLine(GetNumber(new int[] { 1, 2, 3, 5, 10 }, Plus));
    Console.ReadKey();
}
```

- Výsledok je 21 – všetky čísla v poli sa sčítali, nakoľko sme využili metódu `Plus`
- Pozrite si video pre podrobnejšie vysvetlenie

Delegát + anonymná metóda

- Namiesto `Plus` môžeme zavolať **anonymnú metódu**, ktorá je odovzdaná do `GetNumber` a zodpovedá delegátu `MathOperation` a preto ju môžeme použiť:

```
static void Main(string[] args)
{
    //metoda(Plus);
    Console.WriteLine(GetNumber(new int[] { 1, 2, 3, 5, 10 }, delegate (int a, int b) { return a * b; } ));
    Console.ReadKey();
}
```

Delegát + lambda výraz

- Je to podobný zápis ako na predošlom slajde
- Odovzdávame lambda výraz ako delegát

```
static void Main(string[] args)
{
    //metoda(Plus);
    Console.WriteLine(GetNumber(new int[] { 1, 2, 3, 5, 10 }, (a,b) => a * b ));
    Console.ReadKey();
}
```

Delegáty

- Delegáty vieme definovať aj ako bežné premenné

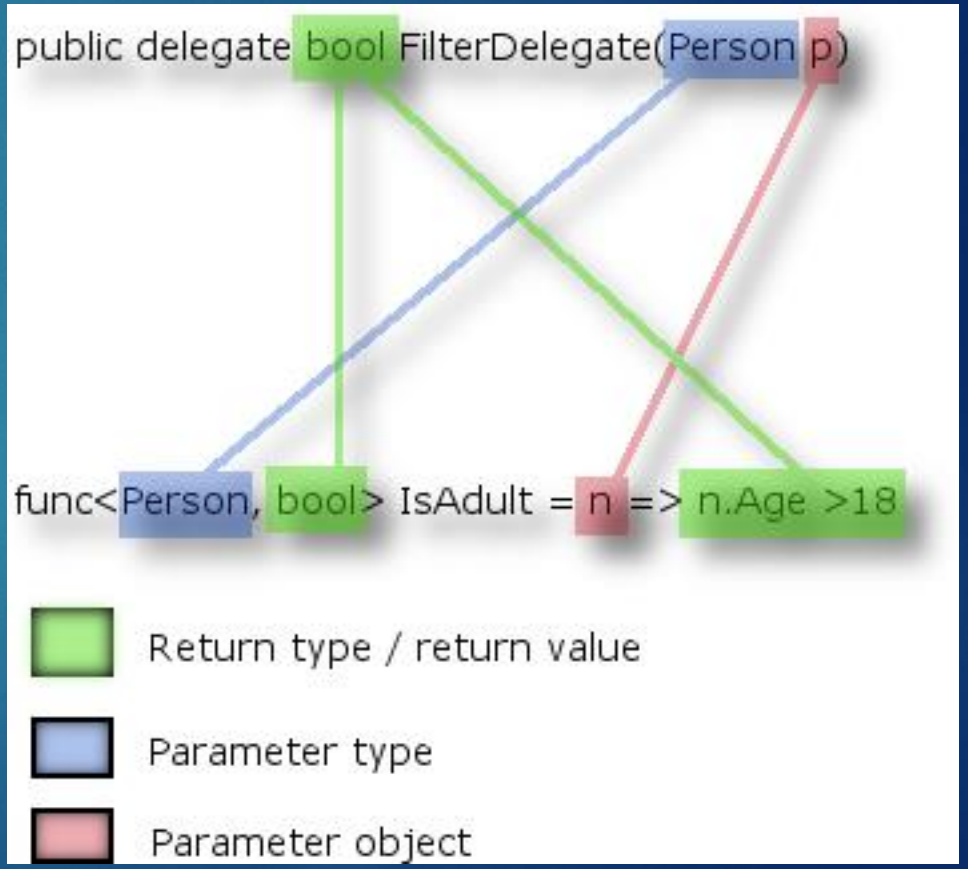
```
delegate int MathOperation(int a, int b);

static void Main(string[] args)
{
    //metoda(Plus);
    MathOperation del = (i, y) => i * y;
    Console.WriteLine(GetNumber(new int[] { 1, 2, 3, 5, 10 }, del ));
    Console.ReadKey();
}
```

- Podobne vieme namiesto lambda výrazu napísať do deklarácie `del` aj anonymnú metódu

Delegáty integrované v .NET

- Ide väčšinou o generické delegáty
- `Func<>`

```
static void Main(string[] args)
{
    Func<int, int, int> del = (a, b) => a * b;
    Console.WriteLine(GetNumber(new int[] { 1, 2, 3, 5, 10 }, del ));
    Console.ReadKey();
}

static int GetNumber(int[] ints, Func<int, int, int> operation)
{
    int result = ints[0];

    for (int i = 1; i < ints.Length; i++)
    {
        result = operation(result, ints[i]);
    }

    return result;
}
```



Lambda výrazy

ERIK KUČERA

PROGRAMOVANIE GUI | PREDNÁŠKA (TÝŽDEŇ 3)

Lambda výraz

- Je to anonymná metóda bez modifikátora prístupu, bez mena, bez slova „return“

Introduction to Lambda Expressions

Introduced with .NET 3.5/C# 3.0

Supersede anonymous methods

Anonymous methods enable you to omit the parameter list

Unnamed, inline functions

Used wherever delegates are required

(input parameters) => expression

Lambda výrazy - vysvetlenie

- Predstavme si, že máme metódu, ktorá nám vráti druhú mocninu čísla
- Zápis pomocou lambda výrazu a delegátov:

```
class Program
{
    static void Main(string[] args)
    {
        // argumenty => vyraz
        // number => number * number;

        Func<int, int> square = number => number*number;

        Console.WriteLine(square(5));
    }

    static int Square (int number)
    {
        return number * number;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // argumenty => vyraz
        // number => number * number;

        Func<int, int> square = Square;

        Console.WriteLine(square(5));
        Console.ReadKey();
    }

    static int Square (int number)
    {
        return number * number;
    }
}
```

Lambda výrazy – iný príklad

```
class Program
{
    static void Main(string[] args)
    {
        // args => expression

        // () => ...
        // x => ...
        // (x, y, z) => ...

        const int factor = 5;

        Func<int, int> multipler = n => n*factor;

        var result = multipler(10);

        Console.WriteLine(result);

    }
}
```

Lambda – praktické použitie - video

Udalosti / events

ERIK KUČERA

PROGRAMOVANIE GUI | PREDNÁŠKA (TÝŽDEŇ 3)

Eventy

- Udalosti alebo eventy sú mechanizmom pre komunikáciu medzi objektmi
- Predstavme si, že máme funkciu na enkódovanie videa, ktorá nám má zaslať mailovú správu, keď enkódovanie skončí
- Takúto funkciu chceme doplniť o funkcionalitu zaslania aj SMS správy
- Riešenie nižšie je **neefektívne**, nakoľko pri každom novom type správy musíme pridať nový riadok kódu

```
public class VideoEncoder
{
    public void Encode(Video video)
    {
        // Encoding logic
        // …

        _mailService.Send(new Mail());

        _messageService.Send(new Text());
    }
}
```

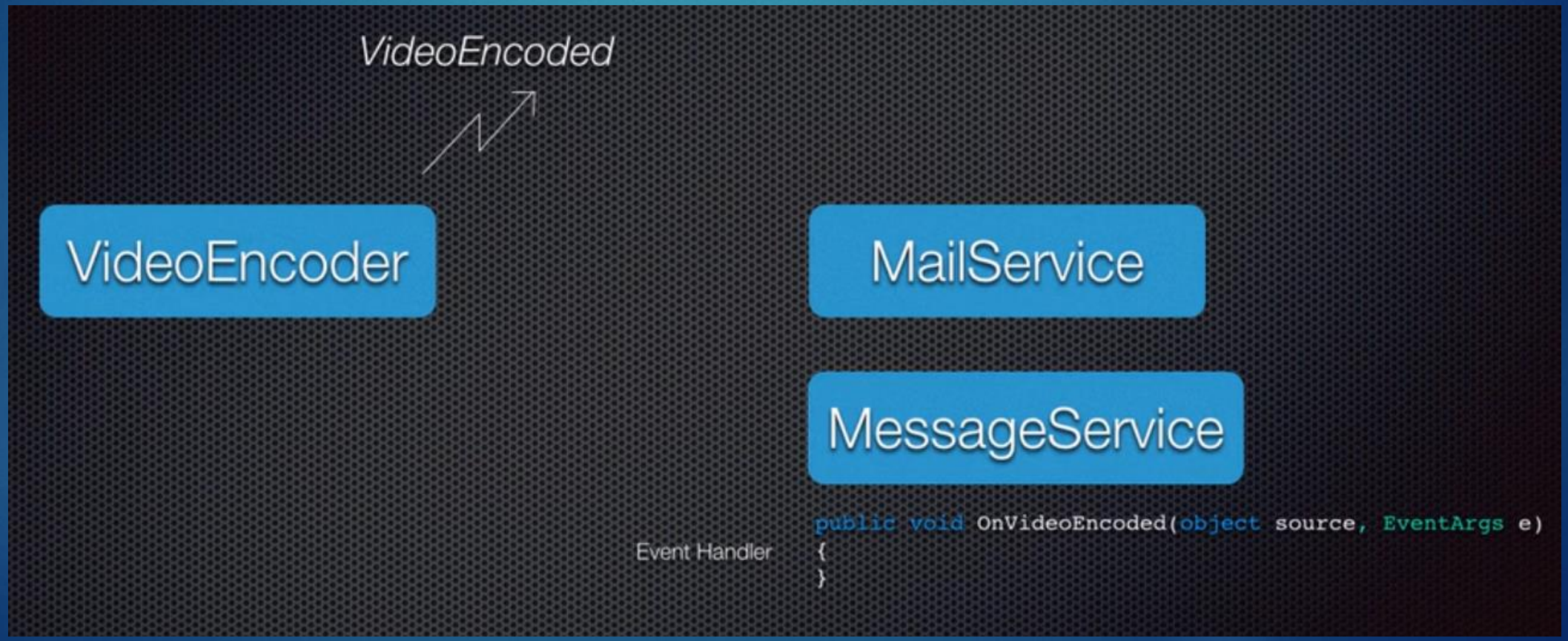
Eventy

- Efektívnejšie je vyslať správu, ktorú prijmú objekty, ktoré si *subscribli* (majú nastavené) prijímanie takýchto správ

```
public class VideoEncoder
{
    public void Encode(Video video)
    {
        // Encoding logic
        // …

        OnVideoEncoded();
    }
}
```

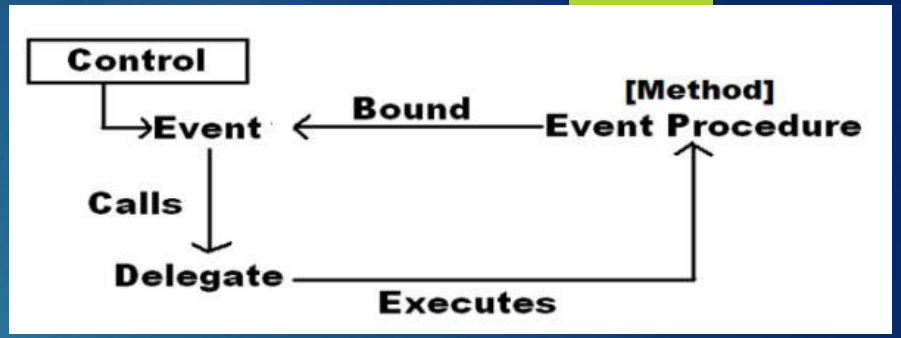
Eventy

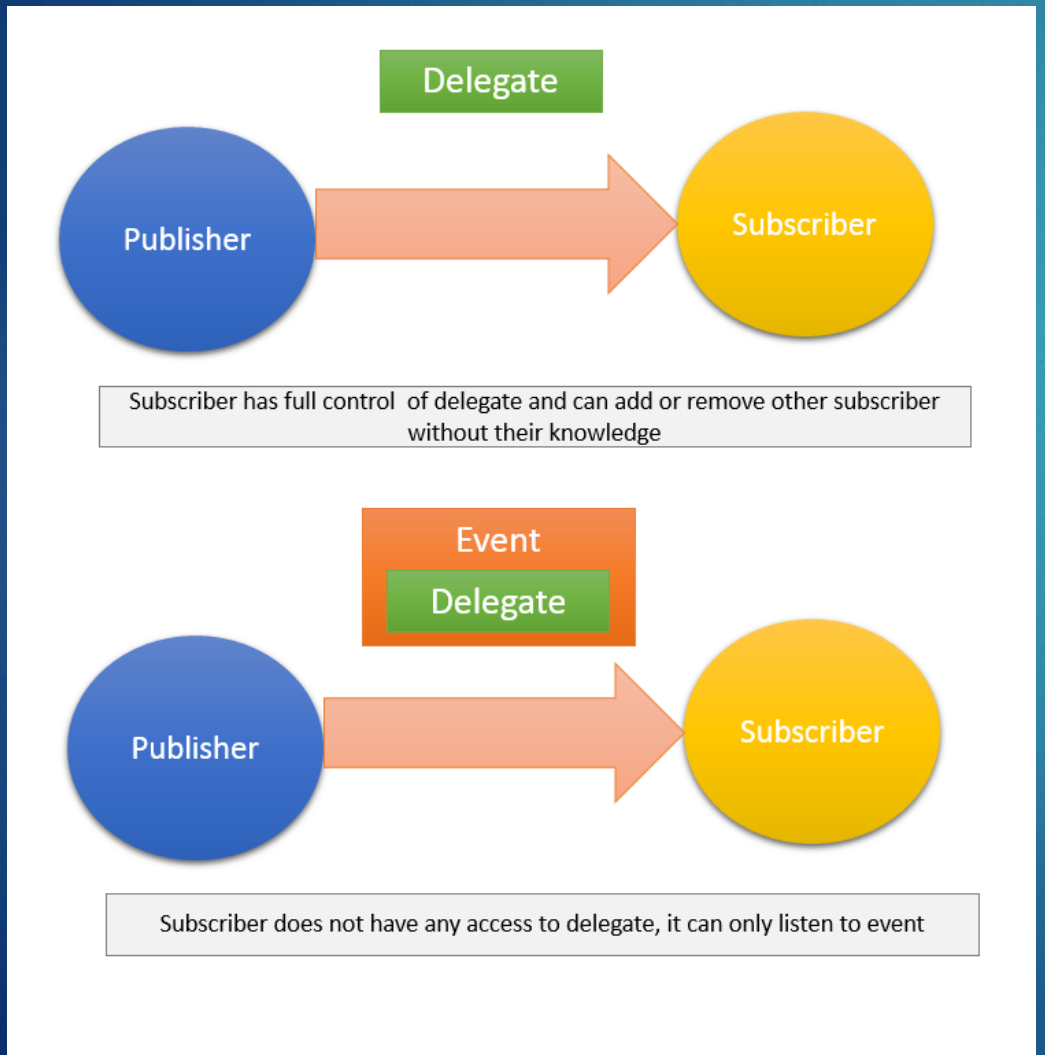
- Vidíme, že objekty na odosielanie mailov a SMS správ obsahujú obslužnú metódu pre správy, ktoré vysiela video enkóder



Delegát v eventoch

- Delegát v tomto prípade funguje ako kontrakt medzi vysielateľom a prijímateľom (subscriberom) správy





Ako naprogramovať spomínanú aplikáciu s eventmi?

- Enkódovanie videa simulujeme nasledovným kódom:

```
public class VideoEncoder
{
    public void Encode(Video video)
    {
        Console.WriteLine("Encoding Video...");
        Thread.Sleep(3000);
    }
}
```

- V hlavnej metóde vytvárame inštanciu videa a videoenkódera:

```
static void Main(string[] args)
{
    var video = new Video() { Title = "Video 1" };
    var videoEncoder = new VideoEncoder();

    videoEncoder.Encode(video);
}
```

Ako naprogramovať spomínanú aplikáciu s eventmi?

- Vo videoenkóderi implementujeme event, pričom musíme dodržať 3 kroky:
  - Definovať delegát
  - Definovať event (udalosť) založenú na delegáte v predošlom bode
  - Spustiť udalosť

```
public class VideoEncoder
{
    // 1- Define a delegate
    // 2- Define an event based on that delegate
    // 3- Raise the event

    public delegate void VideoEncodedEventHandler(object source, EventArgs args);

    public event VideoEncodedEventHandler VideoEncoded;

    public void Encode(Video video)
    {
        Console.WriteLine("Encoding Video...");
        Thread.Sleep(3000);

        OnVideoEncoded();
    }

    protected virtual void OnVideoEncoded()
    {
        if (VideoEncoded != null)
            VideoEncoded(this, EventArgs.Empty);
    }
```

Ako naprogramovať spomínanú aplikáciu s eventmi?

- Následne potrebujeme nejakých subscriberov, ktorí budú na udalosť reagovať

```
class Program
{
    static void Main(string[] args)
    {
        var video = new Video() { Title = "Video 1" };
        var videoEncoder = new VideoEncoder(); //publisher
        var mailService = new MailService(); //subscriber

        videoEncoder.VideoEncoded += mailService.OnVideoEncoded;

        videoEncoder.Encode(video);
    }
}

public class MailService
{
    public void OnVideoEncoded(object source, EventArgs e)
    {
        Console.WriteLine("MailService: Sending an email...");
    }
}
```

Eventy a delegáty v praxi - video

Použité zdroje

- http://wikipedia.org
- http://www.zajtra.sk/programovanie/165/objektovo-orientovane-programovanie-v-normalnej-ludskej-reci
- Syncfusion e-books
- Programming with Mosh
- http://youtube.com