Rozhraní (Interfaces)

Podobně jako abstraktní třídy slouží k abstrakci, tj. definici toho, co mají naše třídy a objekty umět (ne jak to mají dělat!). Narozdíl od abstraktní třídy v rozhraní mohou být jen metody (veřejné) a properties (což, jak už víme, není vlastně proměnná ale syntactic sugar za metodu GetPromenna), žádné proměnné. Rozhraní ani neobsahuje implementaci, všechny položky v něm jsou defaultně `abstract` a `public`. K rozhraní se nepíše (nesmí psát) konstruktor a nejdou vytvářet přímo z něj objekty.

Výhoda oproti abstraktním třídám je, že jedna třída může implementovat více rozhraní (nemůže ale dědit z více tříd!).

Syntax:

```
interface IMojeRozhraní
{



}

class MojeTridaCoImplementujeMojeRozhrani : IMojeRozhrani, ITrebaDalsi
{



}
```

Lambda výrazy

K čemu slouží? Když chcete definovat kus kódu, který ale nepoužijete nikde jinde než právě na tom jednom místě a je dost jednoduchý. Vytvoříte tedy "anonymní metodu", metodu, kterou si sami nepojmenujete.

Lambda výraz má dvě části oddělené znakem `=>` . Před šipkou jsou parametry metody a za šipkou je, co se s nimi má dělat.

Např metoda

```
int AddTwo(int cislo)
{

        return cislo + 2;
}
```

se dá zapsat jako

```
x => x+2;
```

Lambda výrazy se používají například v knihovně `System.Linq`, která nabízí práci s kolekcemi (třeba listy) podobnou jazyku SQL.
Například lze najít všechny prvky listu splňující nějakou podmínku (FIndAll je právě Linq metoda):

```
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0); //najdi suda
```

Čtení ze souboru

Při čtení ze souboru používáme knihovnu `System.IO`. Můžete načíst a zapsat celý soubor najednou:

```
using System.IO;

string textSouboru = File.ReadAllText("jmenosouboru.txt");
string kZapsani = "zapis tohle do souboru."
File.WriteAllText("jmenosouboru.txt",kZapsani);
```

Tohle ale není ideální pro intenzivní práci se souborem, protože to vždy znova projde celý soubor při načtení, uloží do proměnné a zavře (podobně zápis). Nové otevření trvá dlouho, navíc je možné, že se nám soubor nevejde do paměti. Pokud ho chceme procházet po řádce, tak se práce děje dvakrát (poprvé to udělá C# v metodě ReadAllText a podruhé to napíšeme sami). Lepší je otevřít si stream (Příklad použití z dokumentace):

```
string path = @"c:\temp\MyTest.txt";
if (!File.Exists(path))
{
// Create a file to write to.
using (StreamWriter sw = File.CreateText(path))
        {
                sw.WriteLine("Hello");
            sw.WriteLine("And");
            sw.WriteLine("Welcome");
        }
}

        // Open the file to read from.
        using (StreamReader sr = File.OpenText(path))
```

```
        {
            string s;
            while ((s = sr.ReadLine()) != null)
            {
                Console.WriteLine(s);
            }
        }
```

Předchozí kód dělá to, že soubor zavře až na konci bloku `using` , všechno so se děje uvnitř se děje s otevřeným souborem v paměti a práce je rychlá.

Výjimky

Zpracování výjimek

Ideálně by se nemělo stát, že váš program skončí tak, že uživateli spadne a vyhodí výjimku ⟹ měli byste je odchytávat a řešit. Ne obalit celý program do try-catch, který výjimku zahodí. Když už, tak ukázat uživateli nějaké hezké okýnko s tím, že byla vyhozena výjimka. Předvídatelné výjimky byste ale měli řešit sami a nebo psát program tak, aby nevznikali (typicky když píšete metodu na dělení, tak si rozmyslet co se stane při dělení nulou.)

Pozor, program uzavřený v try-catch bloku může být pomalý!

```
try
{
        //some troublemaker code here
}
catch
{
        //what to do, if exception occured
}
```

Je možné také specifikovat různé typy odchytávaných výjimek, použije se ten catch blok, který první (v pořadí odzhora) vyhovuje, tj:

```
try
{
        //some troublemaker code here
}
catch (Exception e)
{
        Console.WriteLine("1");
```

```
}
catch (DivideByZeroException e)
{
        Console.WriteLine("2");
}
finally
{
        //TENTO blok se provede vzdy
}
```

vypíše vždycky 1, protože všechny výjimky jsou potomky třídy Exception.

Vyhazování výjimek

Výjimky můžete i vyhazovat a to vlastní nebo už existující:

```
public int MojeMetoda(int a)
{
        if(a == 0)
        {
                throw new DivideByZeroException("tady je nějaká useful
message");
        }
}
```

Výjimky se dají taky vnořovat, takže ve vašem try-catch bloku vyhodíte novou výjimku, s novou správou a původní vyhozenou výjimkou:

```
try
{
        //some troublemaker code here
}
catch (Exception e)
{
        throw new Exception("Chybne volani moji skvele metody",e);
}
```