Platforma .NET

6. seminář – založeno na materiálech Jiřího Valůška

Radek Janoštík

Univerzita Palackého v Olomouci

30. 10. 2025

Reakce na úkoly

- Většinou využity záznamy (records)
- Odčítání a dělení
- Stromy většinou v pořádku.

F# – Opakování

- Functional-first programovací jazyk
- Ale podpora i ostatních paradigmat (OOP, imperativní)
  - ale snaha se tomu vyhýbat (pokud to jde)
- Od .NET Core je multiplatformní
- Silně typový
- Snaha o stručnost, omezení „kódovacího šumu“ (odvozování typů)
- Čistota a správnost – typový systém, nemutovatelnost
- Jednoduchá paralelizace
- Interoperabilita s jinými jazyky (vs. čistota)

Mutovatelné proměnné a přiřazení

- F# umožňuje vytvořit (klasické) mutovatelné proměnné
- Narušují čisté funkcionální paradigma

```
let x = 5
let mutable y = 10

let addX number = x + number
let addY number = y + number

addX 5
let x = 25
addX 5
// Co bude vysledkem?

addY 10
y <- 1000;;
addY 10;;
// Co bude vysledkem?
```

For cykly

- For-in cyklus (ekvivalent for-each)

```
for pattern in enumerable-expression do
    // telo
```

- Příklad:

```
for i in 1 .. 10 do
    printf "%d " i

for i in 10 .. -2 .. 1 do
    printf "%d " i

for (a,b) in [(1,1); (1,3); (3,2); (4,3); (5,5)] do
    printf "(%d %d) " a b
```

For cykly

- For-to cyklus (klasický cyklus přes řídící proměnnou)

```
for identifier = start [to | downto] finish do
// Telo cyklu
```

- Příklad

```
for i = 1 to 10 do
    printf "%d " i

for i = 10 downto 1 do
    printf "%d " i
```

While cyklus

```
while expr do
    ... // telo cyklu
```

- Příklad

```
let mutable x = 5
while x > 0 do
    printf "%d " x
    x <- x - 1
```

Pole

- = mutovatelné kolekce stejných datových typů s přístupem „přes index“
- Vytvoření

```
let array1 = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
let array2 = [|1 .. 10|]
let array3 = [|'a' .. 'f'|]
```

- Případně funkcemi z modulu Array:

```
let a : int array = Array.zeroCreate 10 // vychozi hodnoty
Array.create 10 "retezec"
let tretiMocniny = Array.init 20 (fun x -> x*x*x)
```

Pole – přístup k prvkům

- Klasicky přes index pomocí `[]`

```
tretiMocniny[0]
```

- Možno získat i podpole pomocí rozsahu (vytvoří kopii)

```
tretiMocniny[2..5]
```

- Mutovatelnost:

```
tretiMocniny[0] <- -9
```

Knihovny .NET

- V F# můžeme používat jakoukoliv .NET knihovnu
- Opět lehce znečišťuje funkcionální přístup

```
open System

let x = 100
let y = "metru"

Console.WriteLine("Delka: {0} {1}", x, y)
```

- Můžeme využít pro mutovatelné kolekce

```
open System.Collections.Generic

let myList = List<int>()
myList.Add(1)
myList.Add(5)
myList[1]
```

Výjimky

- Každý jazyk nad CIL musí mít podporu výjimek
- Snadné nadefinování vastního typu, v catch možný Pattern matching

```
exception MyBigNumberException of string * int * int
let safeDiv x y =
    try
        if x > 5 then
            raise (MyBigNumberException("Velka cisla neumi delit!", x, y))
        else
        Some(x / y)
    with
    | :? System.DivideByZeroException -> printfn "Deleni nulou!"; None
    | MyBigNumberException(text, num,  denum) -> printfn "%s %i %i" text x y
        ; None

let result = safeDiv 10 20
```

Výjimky

- `try-with` blok nelze doplnit finally
- Samostatný blok `try-finally`
- Chceme-li využít oba současně, musíme zanořit

```
exception InnerError of string
exception OuterError of string

let function1 x y =
   try
     try
        if x = y then raise (InnerError("inner"))
        else raise (OuterError("outer"))
     with
      | InnerError(str) -> printfn "Error1 %s" str
   finally
      printfn "Always print this."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

OOP – definice třídy

```
type User(id : int, firstName : string, lastName : string) =
    do // funkce, ktera se provede po primarnim konstruktoru
        printfn "Inicializovano"
    let fullName = firstName + " " + lastName // private promenna
    let mutable access = false // mutable private promenna
    new() = User(0, "John", "Doe") // konstruktor bez parametru
    member this.Id = id // public read-only promenna
    member this.Access // getter a setter
        with get() = access
        and set(value) = access <- value

    member this.SayName = printfn "%s" fullName // public metoda
```

- Pozor na pořadí do/let/new
- Možnost self identifikátorů:
  - `as self` do prvního řádku (scope v celé třídě)
  - `this` na posledním řádku (scope pouze v metodě)

OOP – vytvoření instance a volání metod

- Instance vytvoříme skoro stejně, jako v C#

```
let user = new User(1,"Jan","Novak")
```

- Metody voláme „přes tečku“

```
user.SayName
```

OOP – dědičnost

- („nečekaně“) pouze jednoduchá dědičnost
- Doporučuje se nenadužívat a využít discriminated union

```
type Person(name) =
   member this.Name = name
   member this.Greet() = printfn "Ahoj, jmenuji se %s" this.Name

type Student(name, studentID : int) =
   inherit Person(name)
   let mutable _GPA = 0.0
   member this.StudentID = studentID
   member this.GPA
       with get() = _GPA
       and set value = _GPA <- value
```

OOP – rozhraní

- Definice rozhraní:

```
type IPrintable =
    abstract member Print : unit -> unit
```

- Třída implemetující rozhraní

```
type SomeClass(x: int, y: float) =
    interface IPrintable with
        member this.Print() = printfn "%d %f" x y
```

- Metody z rozhraní nemůžeme volat přímo „přes tečku“ (nepatří dané třídě)
- Musíme volat „přes rozhraní“

```
let s = new SomeClass(1,1.0)
(s :> IPrintable).Print()
```

- Můžeme vytvořit *member*, který volání „přemostí“

```
member this.Print() = (this :> IPrintable).Print()
```

Organizace kódu

- Namespace
  - Kompilují se do .NET `namespace`
  - Nemůže obsahovat definici funkcí, ani hodnoty
  - Mohou být přes více souborů, nezanořují se
- Moduly
  - Common language runtime (CLR) class se statickými metodami
  - Pouze v jednom souboru
  - Mohou se zanořovat
  - Načítáme klíčovým slovem `open název modulu`

```
module Arithmetic // top-level modul
let add x y = x + y
let sub x y = x - y
```

Více modulů

```
module MyModule1 =
    let module1Value = 100
    let module1Function x =
        x + 10

module MyModule2 =
    let module2Value = 121
    let module2Function x =
       x * (MyModule1.module1Function module2Value)
```

Interoperabilita s C#

- Vytvoříme F# knihovnu s modulem

```
module Arithmetic
let add x y = x + y
let sub x y = x - y

let doNothing = fun () -> printf "Nothing from F\#"

let rec fac = function
    | 0 | 1 -> 1
    | n -> n * fac (n - 1)
```

- Přidáme projekt pro C# konzolovou aplikaci
- Přidáme referenci na F# knihovnu (Add→Project reference)
- Funkce najdeme jako statické metody

Úkol

- Vytvořte modul MySort jako knihovnu F#, kterou budete volat z konzolové aplikace v C#.
  - Vytvořte funkcionální a imperativní verzi třídícího algoritmu QuickSort:

```
let quickSortFun (l:int list)
let quickSortImp (arr:int array)
```

  - Obě funkce zavolejte ze C# kódu
- V jazyce F# naprogramujte třídu `Set` ze třetího semináře