Объектно-ориентированное программирование в F#

Юрий Литвинов

18.03.2016г

"За"и "против"ООП в функциональных языках

За:

- Портирование существующего кода
- Интеграция с другими языками
- Использование в основном для ООП с возможностью писать красивый код

Против:

- Не очень дружит с системой вывода типов
- Нет встроенной поддержки печати, сравнения и т.д.

Методы у типов

F#

```
type Vector = {x : float; y : float} with
    member v.Length = sqrt(v.x * v.x + v.y * v.y)

let vector = {x = 1.0; y = 1.0}
let length = vector.Length

type Vector with
    member v.Scale k = {x = v.x * k; y = v.y * k}

let scaled = vector.Scale 2.0
```

Методы у Discriminated Union-ов

F#

```
type Tree<'a> =
    | Tree of 'a * Tree<'a> * Tree<'a>
    | Tip of 'a
    with
        member t.Size =
            match t with
            | Tree(_, l, r) -> 1 + l.Size + r.Size
            | Tip _ -> 1
```

Расширения

F#

```
type System.Int32 with
    member i.IsPrime =
        let limit = i |> float |> sqrt |> int
        let rec check j =
            j > limit or (i % j <> 0 && check (j + 1))
        check 2

printfn "%b" (5).IsPrime
printfn "%b" (8).IsPrime
```

Статические методы

F#

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let vector = Vector.Create 1.0 1.0

type System.Int32 with
    static member IsEven x = x % 2 = 0

printfn "%b" <| System.Int32.IsEven 10
```

Методы и существующие функции

F#

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let length (v : Vector) = sqrt(v.x * v.x + v.y * v.y)

type Vector with
    member v.Length = length v

printfn "%f" <| (Vector.Create 1.0 1.0).Length
printfn "%f" <| (length (Vector.Create 1.0 1.0))
```

Методы и каррирование

F#

```
open Operators

type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let transform v rotate scale =
    let r = System.Math.PI * rotate / 180.0
    { x = scale * v.x * cos r - scale * v.y * sin r;
      y = scale * v.x * sin r + scale * v.y * cos r }

type Vector with
    member v.Transform = transform v

printfn "%A" <| (Vector.Create 1.0 1.0).Transform 45.0 2.0
```

Каррирование против кортежей

F#

```
type Vector with
    member v.TupledTransform (r, s) = transform v r s
    member v.CurriedTransform r s = transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.CurriedTransform 45.0 2.0
```

Кортежи: именованные аргументы

F#

```
type Vector with
    member v.TupledTransform (r, s) =
        transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (r = 45.0, s = 2.0)
printfn "%A" <| v.TupledTransform (s = 2.0, r = 45.0)
```

Кортежи: опциональные параметры

F#

```
type Vector with
    member v.TupledTransform (r, ?s) =
        match s with
        | Some scale -> transform v r scale
        | None -> transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

defaultArg

F#

```
type Vector with
    member v.TupledTransform (r, ?s) =
        transform v r <| defaultArg s 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Кортежи: перегрузка

F#

```
type Vector with
    member v.TupledTransform (r, s) =
        transform v r s
    member v.TupledTransform r =
        transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Кортежи против каррирования

За:

- Можно вызывать из .NET-кода
- Опциональные и именованные аргументы, перегрузки

Против:

- Не поддерживают частичное применение
- Не дружат с функциями высших порядков

Методы против свободных функций

Вывод типов

F#

```
type Vector = {x : float; y : float} with
    member v.Length = v.x * v.x + v.y * v.y |> sqrt

let length v = v.x * v.x + v.y * v.y |> sqrt

let compareWrong v1 v2 =
    v1.Length < v2.Length

let compareRight v1 v2 =
    length v1 < length v2
```

Методы против свободных функций

Функции высших порядков

F#

```
type Vector = {x : float; y : float} with
    member v.Length = v.x * v.x + v.y * v.y |> sqrt

let length v = v.x * v.x + v.y * v.y |> sqrt

let lengths1 = [{x = 1.0; y = 1.0}; {x = 2.0; y = 2.0}]
               |> List.map (fun x -> x.Length)

let lengths2 = [{x = 1.0; y = 1.0}; {x = 2.0; y = 2.0}]
               |> List.map length
```

Классы, основной конструктор

F#

```
type Vector(x, y) =
    member v.Length = x * x + y * y |> sqrt

printfn "%A" <| Vector (1.0, 1.0)
```

F# Interactive

```
FSI_0003+Vector
type Vector =
  class
    new : x:float * y:float -> Vector
    member Length : float
  end
val it : unit = ()
```

Методы и свойства

F#

```
type Vector(x : float, y : float) =
    member v.Scale s = Vector(x * s, y * s)
    member v.X = x
    member v.Y = y
```

F# Interactive

```
type Vector =
  class
    new : x:float * y:float -> Vector
    member Scale : s:float -> Vector
    member X : float
    member Y : float
  end
```

Private-поля и private-методы

F#

```
type Vector(x : float, y : float) =
    let mutable mX = x
    let mutable mY = y
    let lengthSqr = mX * mX + mY * mY
   member v.Length = sqrt lengthSqr
   member v.X = mX
   member v.Y = mY
   member v.SetX x = mX <- x
   member v.SetY y = mY <- y
```

Мутабельные свойства

F#

```
type Vector(x, y) =
    let mutable mX = x
    let mutable mY = y
    member v.X
        with get () = mX
        and set x = mX <- x
    member v.Y
        with get () = mY
        and set y = mY <- y
```

Автоматические свойства

F#

```
type Vector(x, y) =
    member val X = x with get,set
    member val Y = y with get,set

let v = Vector(1.0, 1.0)
v.X <- 2.0
```

Индексеры

F#

```
open System.Collections.Generic

type SparseVector(items : seq<int * float>) =
    let elems = new SortedDictionary<_, _>()
    do items |> Seq.iter (fun (k, v) -> elems.Add(k, v))

    member t.Item
        with get(idx) =
            if elems.ContainsKey(idx) then elems.[idx]
            else 0.0

let v = SparseVector [(3, 547.0)]
printfn "%f" v.[4]
```

Операторы

F#

```
type Vector(x : float, y : float) =
    member v.X = x
    member v.Y = y

    static member (+) (v1 : Vector, v2 : Vector) =
        Vector(v1.X + v2.X, v1.Y + v2.Y)

    static member (-) (v1 : Vector, v2 : Vector) =
        Vector (v1.X - v2.X, v1.Y - v2.Y)

let v = Vector (1.0, 1.0) + Vector (2.0, 2.0)
```

Вернёмся к конструкторам

Дополнительное поведение

F#

```
type Vector(x : float, y : float) =
    let length () = x * x + y * y |> sqrt
    do
        printfn "Vector (%f, %f), length = %f"
            x y <| length ()
        printfn "Have a nice day"
    let mutable x = x
    let mutable y = y

let v = Vector(1.0, 1.0)
```

let-функции и методы

F#

```
type Vector(x : float, y : float) =
    let length () = x * x + y * y |> sqrt
    let normalize () = Vector(x / length(), y / length())
    member this.Normalize = normalize
    member this.X = x
    member this.Y = y

let v = Vector(2.0, 2.0)
let v' = v.Normalize ()
```

Рекурсивные методы

F#

```
type Math() =
    member this.Fibonacci x =
        match x with
        | 0 | 1 -> 1
        | _ -> this.Fibonacci (x - 1)
                + this.Fibonacci (x - 2)

let math = new Math()
printfn "%i" <| math.Fibonacci 10
```

Много конструкторов

F#

```
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y
    new () =
        printfn "Constructor with no parameters"
        Vector(0.0, 0.0)

let v = Vector(2.0, 2.0)
let v' = Vector()
```

Модификаторы видимости

F#

```
type Example() =
    let mutable privateValue = 42

    member this.PublicValue = 1
    member private this.PrivateValue = 2
    member internal this.InternalValue = 3

    member this.PrivateSetProperty
        with get () =
            privateValue
        and private set(value) =
            privateValue <- value
```

Наследование

F#

```
type Shape() =
    class
    end

type Circle(r) =
    inherit Shape()
    member this.R = r
```

Абстрактные классы

F#

```
[<AbstractClass>]
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    inherit Shape()
    member this.R = r
    override this.Draw () =
        printfn "Drawing circle"
    override this.Name = "Circle"
```

Реализация по умолчанию

F#

```
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string
    default this.Draw () =
        printfn "Drawing shape"
    default this.Name =
        "Shape"
```

Вызов метода родителя

F#

```
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string
    default this.Draw () = printfn "Drawing shape"
    default this.Name = "Shape"

type Circle(r) =
    inherit Shape()
    member this.R = r
    override this.Draw () =
        base.Draw ()
        printfn "Drawing circle"
    override this.Name = "Circle"
```

Интерфейсы

F#

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    member this.R = r
    interface Shape with
        member this.Draw () =
            printfn "Drawing circle"
        member this.Name = "Circle"
```

Явное приведение типов

F#

```
let c = Circle 10
c.Draw ()   // Ошибка
(c :> Shape).Draw ()   // Ок

let draw (s : Shape) = s.Draw ()

draw c   // Ок
```

Наследование интерфейсов

F#

```
type IEnumerable<'a> =
    abstract GetEnumerator : unit -> IEnumerator<'a>

type ICollection<'a> =
    inherit IEnumerable<'a>
    abstract Count : int
    abstract IsReadOnly : bool
    abstract Add : 'a -> unit
    abstract Clear : unit -> unit
    abstract Contains : 'a -> bool
    abstract CopyTo : 'a[] * int -> unit
    abstract Remove : 'a -> unit
```

Объектные выражения

Реализация интерфейсов на лету

F#

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

let rect w h =
    { new Shape with
        member this.Draw () =
            printfn "Drawing rect, w = %d, h = %d" w h
        member this.Name = "Rectange"
    }

(rect 10 10).Draw ()
```

Частичная реализация интерфейса

F#

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

let simpleShape nameFunc =
    { new Shape with
        member this.Draw () =
            printfn "Drawing %s" this.Name
        member this.Name = nameFunc ()
    }

(simpleShape (fun () ->"Star")).Draw ()
```

Делегация вложенному классу

F#

```
type Printer =
    abstract member WriteString : string -> unit

type HtmlWriter() =
    let mutable count = 0
    let printer =
        { new Printer with
            member this.WriteString s =
                count <- count + s.Length
                System.Console.Write(s) }
    member x.CharCount = count
    member x.Header () = printer.WriteString "<html>"
    member x.Footer () = printer.WriteString "</html>"
    member x.WriteString s = printer.WriteString s
```

Модули

F#

```
type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
    let scale k v = { x = k * v.x; y = k * v.y }
    let shiftX x v = { v with x = v.x + x }
    let shiftY y v = { v with y = v.y + y }
    let shiftXY (x, y) v = { x = v.x + x; y = v.y + y }
    let zero = { x = 0.0; y = 0.0 }
    let constX dx = { x = dx; y = 0.0 }
    let constY dy = { x = 0.0; y = dy }
```

Расширения модулей

F#

```
module List =
    let rec pairwise l =
        match l with
        | [] | [_] -> []
        | h1 :: (h2 :: _ as t) -> (h1, h2) :: pairwise t

let x = List.pairwise [1; 2; 3; 4]
```

Дотнетовские структуры

F#

```
[<Struct>]
type VectorStruct =
    val x : float
    val y : float
    new (x, y) = {x = x; y = y}
    member v.X = v.x
    member v.Y = v.y
    member v.Length = v.x * v.x + v.y * v.y |> sqrt

type VectorStruct' =
    struct
        val x: float
        val y: float
    end
```

Пространства имён

F#

```
namespace Vectors

type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
```