عنوان: **شی گرایی در F#**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۲۰ ۲:۱۳
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

برنامه نویسی شی گرای سومین نسل از الگوهای اصلی برنامه نویسی است. در توضیحات فصل اول گفته شد که F# یک زبان تابع گرا است ولی این بدان معنی نیست که F# از مفاهیمی نظیر کلاس و یا interface پشتیبانی نکند. برعکس در F# امکان تعریف کلاس و interface و هم چنین پیاده سازی مفاهیم شی گرایی وجود دارد.

*با توجه به این موضوع که فرض است دوستان با مفاهیم شی گرایی آشنایی دارند از توضیح و تشریح این مفاهیم خودداری می‌کنم.

Classes

کلاس چارچوبی از اشیا است برای نگهداری خواص(Properties) و رفتار ها(Methods) و رخدادها(Events). کلاس پایه ای‌ترین مفهوم در برنامه نویسی شی گراست. ساختار کلی تعریف کلاس در F# به صورت زیر است:

```
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
   [ class ]
     [ inherit base-type-name(base-constructor-args) ]
     [ let-bindings ]
     [ do-bindings ]
     member-list
       ...
   [ end ]

type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

همان طور که در ساختار بالا می‌بینید مفاهیم access-modifier و inherit و constructor هم در F# وجود دارد.

انواع access-modifier در F#
public : دسترسی برای تمام فراخوان‌ها امکان پذیر است
internal : دسترسی برای تمام فراخوان هایی که در همین assembly هستند امکان پذیر است
private : دسترسی فقط برای فراخوان‌های موجود در همین ماژول امکان پذیر است

نکته : protected access modifier در F# پشتیبانی نمی‌شود.

مثالی از تعریف کلاس:

```
type Account(number : int, name : string) = class
    let mutable amount = 0m

end
```

کلاس بالا دارای یک سازنده است که دو پارامتر ورودی می‌گیرد. کلمه end به معنای انتهای کلاس است. برای استفاده کلاس باید به صورت زیر عمل کنید:

```
let myAccount = new Account(123456, "Masoud")
```

توابع و خواص در کلاس ها

برای تعریف خاصیت در F# باید از کلمه کلیدی member استفاده کنید. در مثال بعدی برای کلاس بالا تابع و خاصیت تعریف خواهیم کرد.

```
type Account(number : int, name: string) = class
    let mutable amount = 0m

    member x.Number = number
    member x.Name= name
    member x.Amount = amount

    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount - value
end
```

کلاس بالا دارای سه خاصیت به نام‌های Number و Name و Amount است و دو تابع به نام‌های Deposit و Withdraw دارد. اما x استفاده شده قبل از هر member به معنی this در C# است. در #F شما برای اشاره به شناسه‌های یک محدوده خودتون باید یک نام رو برای اشاره گر مربوطه تعیین کنید.

```
open System

type Account(number : int, name: string) = class
    let mutable amount = 0m

    member x.Number = number
    member x.Name= name
    member x.Amount = amount

    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount - value
end



 let masoud= new Account(12345, "Masoud")
 let saeed = new Account(67890, "Saeed")

let transfer amount (source : Account) (target : Account) =
    source.Withdraw amount
    target.Deposit amount

let printAccount (x : Account) =
    printfn "x.Number: %i, x.Name: %s, x.Amount: %M" x.Number x.Name x.Amount

let main() =
    let printAccounts() =
        [masoud; saeed] |> Seq.iter printAccount

    printfn "\nInializing account"
    homer.Deposit 50M
    marge.Deposit 100M
    printAccounts()

    printfn "\nTransferring $30 from Masoud to Saeed"
    transfer 30M masoud saeed



    printAccounts()

    printfn "\nTransferring $75 from Saeed to Masoud"
    transfer 75M saeed masoud
    printAccounts()

main()
```

استفاده از کلمه do

در #F زمانی که قصد داشته باشیم در بعد از وهله سازی از کلاس و فراخوانی سازنده، عملیات خاصی انجام شود(مثل انجام برخی عملیات متداول در سازنده‌های کلاس‌های دات نت) باید از کلمه کلیدی do به همراه یک بلاک از کد استفاده کنیم.

```
open System
open System.Net

type Stock(symbol : string) = class

    let mutable _symbol = String.Empty
```

```
    do
     //کد مورد نظر در این جا نوشته  میشود
end
```

یک مثال در این زمینه:

```
open System

type MyType(a:int, b:int) as this =
    inherit Object()
    let x = 2*a
    let y = 2*b
    do printfn "Initializing object %d %d %d %d %d %d"
               a b x y (this.Prop1) (this.Prop2)
    static do printfn "Initializing MyType."
    member this.Prop1 = 4*x
    member this.Prop2 = 4*y
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)

let obj1 = new MyType(1, 2)
```

در مثال بالا دو عبارت do یکی به صورت static و دیگری به صورت غیر static تعریف شده اند. استفاده از do به صورت غیر static این امکان را به ما می‌دهد که بتوانیم به تمام شناسه‌ها و توابع تعریف شده در کلاس استفاده کنیم ولی do به صورت static فقط به خواص و توابع از نوع static در کلاس دسترسی دارد.
خروجی مثال بالا:

```
Initializing MyType.
Initializing object 1 2 2 4 8 16
```

خواص static:
برای تعریف خواص به صورت استاتیک مانند #C از کلمه کلیدی static استفاده کنید.مثالی در این زمینه:

```
type SomeClass(prop : int) = class
    member x.Prop = prop
    static member SomeStaticMethod = "This is a static method"
end
```

SomeStaticMethod به صورت استاتیک تعریف شده در حالی که x.Prop به صورت غیر استاتیک. دسترسی به متدها یا خواص static باید بدون وهله سازی از کلاس انجام بگیرد در غیر این صورت با خطای کامپایلر روبرو خواهید شد.

```
let instance = new SomeClass(5);;
instance.SomeStaticMethod;;

output:
stdin(81,1): error FS0191: property 'SomeStaticMethod' is static.
```

روش استفاده درست:

```
SomeClass.SomeStaticMethod;; (* invoking static method *)
```

متدهای get , set در خاصیت ها:
همانند #C و سایر زبان‌های دات نت امکان تعریف متدهای get و set برای خاصیت‌های یک کلاس وجود دارد.
ساختار کلی:

```
member alias.PropertyName
        with get() = some-value
        and set(value) = some-assignment
```

مثالی در این زمینه:

```
type MyClass() = class
   let mutable num = 0
    member x.Num
        with get() = num
        and set(value) = num <- value
end;;
```

کد متناظر در C#:

```
public int Num
{
   get{return num;}
   set{num=value;}
}
```

یا به صورت:

```
type MyClass() = class
    let mutable num = 0

    member x.Num
        with get() = num
        and set(value) =
            if value > 10 || value < 0 then
                raise (new Exception("Values must be between 0 and 10"))
            else
                num <- value
end
```

Interface ها

اینترفیس به تمامی خواص و توابع عمومی اشئایی که آن را پیاده سازی کرده اند اشاره می‌کند. (توضیحات بیشتر (^) و (^))ساختار کلی برای تعریف آن به صورت زیر است:

```
type type-name =
   interface
       inherits-decl
       member-defns
   end
```

مثال:

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

استفاده از حرف I برای شروع نام اینترفیس طبق قوانین تعریف شده (اختیاری) برای نام گذاری است.

نکته: در هنگام تعریف توابع و خاصیت در interfaceها باید از کلمه abstract استفاده کنیم. هر کلاسی که از یک یا چند تا اینترفیس ارث ببرد باید تمام خواص و توابع اینتریس‌ها را پیاده سازی کند. در مثال بعدی کلاس SomeClass1 اینترفیس بالا را پیاده سازی می‌کند. دقت کنید که کلمه this توسط من به عنوان اشاره گر به اشیای کلاس تعیین شده و شما می‌تونید از هر کلمه یا حرف دیگری استفاده کنید.

```
type SomeClass1(x: int, y: float) =
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y
```

نکته مهم: اگر قصد فراخوانی متد Print را در کلاس بالا دارید نمی‌تونید به صورت مستقیم متد بالا را فراخوانی کنید. بلکه حتما باید کلاس به اینترفیس مربوطه cast شود.
روش نادرست:

```
let instance = new SomeClass1(10,20)
instance.Print//فراخوانی این متد باعث ایجاد خطای کامپایلری می‌شود.
```

روش درست:

```
let instance = new SomeClass1(10,20)
let instanceCast = instance :> IPrintable// استفاده از (:>)  برای عملیات تبدیل کلاس به اینترفیس
instanceCast.Print
```

برای عملیات cast ازاستفاده کنید.
در مثال بعدی کلاسی خواهیم داشت که از سه اینترفیس ارث می‌برد. در نتیجه باید تمام متدهای هر سه اینترفیس را پیاده سازی کند.

```
type Interface1 =
    abstract member Method1 : int -> int

type Interface2 =
    abstract member Method2 : int -> int

type Interface3 =
    inherit Interface1
    inherit Interface2
    abstract member Method3 : int -> int

type MyClass() =
    interface Interface3 with
        member this.Method1(n) = 2 * n
        member this.Method2(n) = n + 100
        member this.Method3(n) = n / 10
```

فراخوانی این متدها نیز به صورت زیر خواهد بود:

```
let instance = new MyClass()
let instanceToCast = instance :> Interface3
instanceToCast.Method3 10
```

کلاس‌های Abstract

#F از کلاس‌های abstract هم پشتیبانی می‌کند. اگر با کلاس‌های abstract در #C آشنایی ندارید می‌تونید مطالب مورد نظر رو در (^) و (^) مطالعه کنید. به صورت خلاصه کلاس‌های abstract به عنوان کلاس‌های پایه در برنامه نویسی شی گرا استفاده می‌شوند. این کلاس‌ها دارای خواص و متدهای پیاده سازی شده و نشده هستند. خواص و متد هایی که در کلاس پایه abstract پیاده سازی نشده اند باید توسط کلاس هایی که از این کلاس پایه ارث می‌برند حتما پیاده سازی شوند.
ساختار کلی تعریف کلاس‌های abstract:

```
[<AbstractClass>]
type [ accessibility-modifier ] abstract-class-name =
    [ inherit base-class-or-interface-name ]
    [ abstract-member-declarations-and-member-definitions ]

    abstract member member-name : type-signature
```

در #F برای این که مشخص کنیم که یک کلاس abstract است حتما باید [<AbstractClass>] در بالای کلاس تعریف شود.

```
[<AbstractClass>]
type Shape(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
```

```
    let mutable rotAngle = 0.0

    abstract Area : float with get
    abstract Perimeter : float  with get
    abstract Name : string with get
```

کلاس بالا تعریفی از کلاس abstract است که سه خصوصیت abstract دارد (برای تعیین خصوصیت‌ها و متد هایی که در کلاس پایه پیاده سازی نمی‌شوند از کلمه کلیدی abstract در هنگام تعریف آن‌ها استفاده می‌کنیم). حال دو کلاس ایجاد می‌کنیم که این کلاس پایه را پیاده سازی کنند.

#1 کلاس اول

```
type Square(x, y,SideLength) =
    inherit Shape(x, y)




   override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"
```

#2 کلاس دوم

```
type Circle(x, y, radius) =
    inherit Shape(x, y)



    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
```

Structures

structureها در #F دقیقا معال struct در #C هستند. توضیحات بیشتر درباره struct در #C (^) و (^)). اما به طور خلاصه باید ذکر کنم که strucureها تقریبا دارای مفهوم کلاس هستند با اندکی تفاوت که شامل موارد زیر است:
structureها از نوع مقداری هستند و این بدین معنی است مستقیما درون پشته ذخیره می‌شوند.
ارجاع به structureها از نوع ارجاع با مقدار است بر خلاف کلاس‌ها که از نوع ارجاع به منبع هستند.(^)
structureها دارای خواص ارث بری نیستند.

عموما از structure برای ذخیره مجموعه ای از داده‌ها با حجم و اندازه کم استفاده می‌شود.

ساختار کلی تعریف structure

```
[ attributes ]
type [accessibility-modifier] type-name =
   struct
      type-definition-elements
   end

//یا به صورت زیر

[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
   type-definition-elements
```

یک نکته مهم هنگام کار با structها در #F این است که امکان استفاده از let و Binding در structها وجود ندارد. به جای آن

باید از val استفاده کنید.

```
type Point3D =
   struct
      val x: float
      val y: float
      val z: float
   end
```

تفاوت اصلی بین val و let در این است که هنگام تعریف شناسه با val امکان مقدار دهی اولیه به شناسه وجود ندارد. در مثال بالا مقادیر برای x و y و z برابر 0.0 است که توسط کامپایلر انجام می‌شود. در ادامه یک struct به همراه سازنده تعریف می‌کنیم:

```
type Point2D =
   struct
      val X: float
      val Y: float
      new(x: float, y: float) = { X = x; Y = y }
   end
```

توسط سازنده struct بالا مقادیر اولیه x و y دریافت می‌شود به متغیرهای متناظر انتساب می‌شود.

در پایان یک مثال مشترک رو در C# و F# پیاده سازی می‌کنیم:

C#

```
abstract class Shape { }
class Line : Shape {
 public Point Pt1;
 public Point Pt2;
}
class Square : Shape {
 public Point Pt;
 public float Size;
}
void Draw(Graphics g,Pen pen,Shape shape)
{
 if (shape is Line)
 {
  var line = (Line)shape;
  g.DrawLin(pen,line.Pt1,line.Pt2);
}
 else if (shape is Square)
{
  var sq = (Square)shape;
  g.DrawRectangle(pen,
   sq.Pt.X, sq.Pt.Y,
   sq.Size, sq.Size);
 }
}
```

F#

```
type Shape =
| Line of Point * Point
| Square of Point * float

let draw (g:Graphics,pen:Pen,shape)=
  match shape with
  | Line(pt1,pt2) ->
     g.DrawLine(pen,pt1,pt2)
  | Square(pt,size) ->
     g.DrawRectangle(pen,
      pt.X,pt.Y,size,size)
```