عنوان: **معرفی #F**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۷ ۲:۲۴
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

یکی از قدیمی‌ترین روش‌های برنامه نویسی روش برنامه نویسی تابع گراست. زبان IPL به عنوان قدیمی‌ترین زبان برنامه نویسی تابع گرا در سال 1955(یک سال قبل از خلق فرترن) است. دومین زبان تابع گرا زبان LISP بوده است که در سال 1958(یک سال قبل از خلق کوبول) متولد شد. هر دو زبان کوبول و فرترن زبان‌های امری و رویه ای بودند. بعد از آن‌ها در سال 1970 شروع عرصه زبان‌های شی گرا بود و تا امروز بیشترین کاربرد را در تولید نرم افزارها داشته اند.
F# یک زبان برنامه نویسی تابع گرا است و گزینه ای بسیار مناسب برای حل مسایل کامپیوتری. اما استفاده از زبان برنامه نویسی تابعی محض برای نوشتن و تولید پروژه‌های نرم افزاری مناسب نمی‌باشد. به همین دلیل نیاز به استفاده از این زبان‌ها در کنار سایر زبان‌های شی گرا احساس می‌شود. F# یک زبان همه منظوره دات نت است که برای حالت اجرا به صورت همه منظوره استفاده می‌شود. برخی زبان‌های تابع گرا دیگر نظیر Lisp و Haskel و OCaml (که F# بسیار نزدیک به این زبان می‌باشد) با دستورات زبان اجرای سفارشی کار می‌کنند و این مسئله باعث نبود زبان برنامه نویسی چند فعالیته می‌شود. شما می‌توانید از برنامه نویسی توصیفی هم استفاده کنید و توابع را به راحتی با هم ترکیب کنید و یا روش‌های شی گرایی و دستوری را در همان برنامه استفاده کنید.

تاریخچه

F# توسط دکتر دون سیم ابداع شد. در حال حاضر F# وابسته به تیمی کوچک ولی پیشرفته واقع در مرکز تحقیقات شرکت مایکروسافت می‌باشد. F#مدل خود را از روی زبان برنامه نویسی OCAML انتخاب کرد و سپس با گسترش قابلیت‌های فنی، خود را در دات نت گنجاند. F# در بسیاری از برنامه‌های بزرگ دنیای واقعی استفاده شده است که این خود نمایانگر آکادمیک نبودن محض این زبان است. با توجه به اینکه زبان تابع گرای دیگر به ندرت در دات نت توسعه پیدا کرده است F# به عنوان استاندارد در این مقوله در آمده است. زبان F# از نظر کیفیت و سازگار بودن با دات نت و VisualStudio بسیار وضعیت بهتری نسبت به رقبای خود دارد و این خود دلیلی دیگری است برای انتخاب این زبان.

استفاده در دات نت

F# کاملا از دات نت پشتیبانی می‌کند و این قابلیت را به برنامه نویسان می‌دهد که هر چیزی را که در سایر زبان‌های دات نت استفاده می‌کنند در این زبان نیز قابل استفاده باشد. همچنین می‌تواند برای کد نویسی IL نیز استفاده شود.
F# به راحتی قابل اجرا در محیط لینوکس و مکینتاش نیز است.

استفاده کنندگان F#

F# در شرکت مایکرو سافت به شدت استفاده می‌شود. رالف هربریش که یکی از مدیران دوگانه گروه بازی‌های مایکروسافت و از متخصصین آموزش ماشین است در این باره می‌گوید:
اولین برنامه کاربردی برای انتقال 110 گیگا بایت از طریق 11000 فایل متنی در بیش از 300 دایرکتوری و وارد کردن آن‌ها در دیتابیس بود. کل برنامه 90 خط بود و در کمتر از 18 ساعت توانست اطلاعات مربوطه را در SQL ذخیره کند. یعنی ده هزار خط برنامه متنی در هر ثانیه مورد پردازش قرار گرفت.همچنین توجه کنید که من برنامه را بهینه نکردم بلکه به صورت کاملا عادی نوشتم. این جواب بسیار قابل توجه بود زیرا من انتظار داشتم حداقل یک هفته زمان ببرد.
دومین برنامه، برنامه پردازش میلیون‌ها Feekback مشتریان بود. ما روابط مدلی زیادی را توسعه دادیم و من این روابط را در F# قرار دادم و داده‌های مربوط به SQL را در آن فراخوانی کردم و نتایج را در فایل داده ای MATLAB قرار دادم و کل پروژه در حد صد خط بود به همراه توضیحات. زمان اجرای پروژه برای دریافت خروجی ده دقیقه بود در حالی که همین کار را توسط برنامه C# قبلا توسعه داده بودیم که بیش از هزار خط بود و نزدیک به دو روز زمان می‌برد.

استفاده از F# تنها در مایکروسافت نیست بلکه در سایر شرکت‌های بزرگ و نام دار نیز استفاده می‌شود و همچنان نیز در حال افزایش است. شرکت
Derivative One که یک شرکت بزرگ در تولید نرم افزارهای شبیه ساز مالی است مدل‌های مالی نرم افزارهای خود را در F#

پیاده سازی کرده است.

چرا #F ؟

همیشه باید دلیلی برای انتخاب یک زبان باشد. در حال حاضر #F یکی از قدرتمندترین زبان‌های برنامه نویسی است. در ذیل به چند تا از این دلایل اشاره خواهم کرد:

#F یک زبان استنباطی است. برای مثال در هنگام تعریف متغیر و شناسه نیاز به ذکر نوع آن نیست. کامپایلر با توجه به مقدار اولیه تصمیم می‌گیرد که متغیر از چه نوعی است.

بسیار راحت می‌توان به کتابخانه قدرتمند دات نت دسترسی داشت و از آن‌ها در پروژه‌های خود استفاده کنید.

#F از انواع روش‌های برنامه نویسی نظیر **تابعی، موازی، شی گرا و دستوری** پیشتیبانی می‌کند.

برخلاف تصور بعضی افراد، در #F امکان تهیه و توسعه پروژه‌های وب و ویندوز و حتی WPF و Silverlight هم وجود دارد.

نوع کدنویسی و syntax زبان #F به برنامه نویسان این اجازه را میدهد که الگوریتم‌های پیچیده مورد نظر خود را بسیار راحت‌تر پیاده سازی کنند. به همین دلیل بعضی برنامه نویسان این زبان را با Paython مقایسه می‌کنند.

#F به راحتی با زبان #C و VB تعامل دارد. یعنی می‌تونیم در طی روند تولید پروژه از قدرت‌های هر سه زبان بهره بگیریم.

طبق آمار گرفته شده از برنامه نویسان، #F به دلیل پشتیبانی از نوع داده ای قوی و مبحث Unit Measure، خطاها و Bugهای نرم افزار را کاهش می‌دهد.

به دلیل پشتیبانی VS.Net از زبان #F و وجود ابزار قدرتمند برای توسعه نرم افزار به کمک این زبان (unitTesting و ابزارهای debuging و ..)این زبان تبدیل به قدرت‌های دنیای برنامه نویسی شده است.

#F یک زبان بسیار مناسب برای پیاده سازی الگوریتم‌های data-mining است.

#F از immutability در تعریف شناسه‌ها پشتیبانی می‌کند.(در فصل‌های مربوطه بحث خواهد شد)

و.....

چرا #F نه ؟

#F هم مانند سایر زبان ها، علاوه بر قدرت بی همتای خود دارای معایبی نیز می‌باشد. (مواردی که در پایین ذکر می‌شود صرفا بر اساس تجربه است نه مستندات).

نوع کدنویسی و syntax زبان #F برای برنامه نویسان دات بیگانه (و البته کمی آزار دهنده) است. اما به مرور این مشکل، تبدیل به قدرت برای مانورهای مختلف در کد می‌شود.

درست است که در #F امکان تعریف اینترفیس وجود دارد و یک کلاس می‌تواند اینترفیس مورد نظر را پیاده سازی کند ولی هنگام فراخوانی متدهای کلاس (اون هایی که مربوط به اینترفیس است) حتما باید instance کلاس مربوطه به اینترفیس cast شود و این کمی آزار دهنده است.(در فصل شی گرایی در این مورد شرح داده شده است).

زبان #F در حال حاضر توسط VS.Net به صورت Visual پشتیبانی نمی‌شود.(امکاناتی نظیر drag drop کنترل‌ها برای ساخت فرم و ....). البته برای حل این مشکل نیز افزونه هایی وجود دارد که در جای مناسب بحث خواهیم کرد.

آیا برای یادگیری #F نیاز به داشتن دانش در برنامه نویسی #C یا VBداریم؟

به طور قطع نه. نوع کد نویسی (نه مفاهیم)در #F کاملا متفاوت در #C است و این دو زبان از نظر کد نویسی شباهتشان در حد صفر است. برای یادگیری #F بیشتر نیاز به داشتن آگاهی اولیه در برنامه نویسی (آشنایی با تابع، حلقه تکرار، متغیر ها) و شی گرایی(مفاهیم کلاس، اینترفیس، خواص، متدها و...) دارید تا آشنایی با #C یا VB.

چگونه شروع کنیم؟

اولین گام برای یادگیری آشنایی با نحوه کد نویسی #F است. بدین منظور در طی فصول آموزش سعی بر این شده است از مثال‌های بسیار زیاد برای درک بهتر مفاهیم استفاده کنم. تا جای ممکن برای اینکه تکرار مکررات نشود و شما خواننده عزیز به خاطر مطالب واضح و روشن خسته نشوید از تشریح مباحث واضح خودداری کردم و بیشتر به پیاده سازی مثال اکتفا نمودم.

نظرات خوانندگان

نویسنده: نریمان
تاریخ: ۱۳۹۲/۰۴/۲۳ ۱۵:۱۶

با تشکر از این مجموعه خیلی خوب. مفاهیم کلی F شارپ رو به طور مناسبی خلاصه کردید.
اما چندتا نکته به نظرم رسید که اگر اصلاح بشه، بهبود قابل توجهی ایجاد می‌شه.
۱- خیلی از اصطلاح‌ها به فارسی برگردانده شده‌اند اما معادل انگلیسی همه آن‌ها ذکر نشده. به نظرم ذکر عبارت انگلیسی آن حتی از استفاده عبارت فارسی مهمتره، چون در نهایت کاربر برای یادگیری بیشتر باید از منابع انگلیسی استفاده کنه. پس چه بهتر که این‌جا یک مقدار آشنا بشه.
۲- کاش جایی که درباره تابع‌گرا بودن صحبت می‌کردید، از ابتدا درباره مفهوم اصلی اون و تفاوتی که بین شی‌گرایی وجود داره بحث می‌شد و اون رو زمان یاددادن نحوه تعریف تابع و ... موکول نمی‌کردید.
یعنی مقداری درباره تفاوتی که شی‌گرایی بین فیلد و متد می‌گذاره و این‌جا با تابع و شناسه از یک جنس برخورد می‌شه (اگه اشتباه می‌کنم لطفا تصحیح بفرمایین)، از نظر ریاضی چند تا مثال می‌زدید تا مفهوم اون جا بیفته.
۳- یه جاهایی لحن متن بین رسمی و محاوره گیر کرده. یعنی یه جایی خیلی رسمی شده، یه جایی نه. اگه یه مقدار یکدست‌تر بنویسید، تمرین نگارشیه خوبیه :دی

عنوان: **شناسه ها و استفاده از Let**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۷ ۳:۳۳
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

F# هم مانند سایر زبان‌های برنامه نویسی از یک سری Data Type به همراه عملگر و Converter پشتیبانی می‌کند که در ابتدا لازم است یک نگاه کلی به این موارد بیندازیم. به دلیل آشنایی اکثر دوستان به این موارد و به دلیل اینکه تکرار مکررات نشود از توضیح در این موارد خودداری خواهم کرد.(در صورت مبهم بودن می‌توانید از قسمت پرسش و پاسخ استفاد نمایید)

Basic Literal

Type	Description	Sample Literals	.NET Name
bool	True/false values	true, false	System.Boolean
byte	8-bit unsigned integers	0uy, 19uy, 0xFFuy	System.Byte
sbyte	8-bit signed integers	0y, 19y, 0xFFy	System.SByte
int16	16-bit signed integers	0s, 19s, 0x0800s	System.Int16
uint16	16-bit unsigned integers	0us, 19us, 0x0800us	System.UInt16
int, int32	32-bit signed integers	0, 19, 0x0800, 0b0001	System.Int32
uint32	32-bit unsigned integers	0u, 19u, 0x0800u	System.UInt32
int64	64-bit signed integers	0L, 19L, 0x0800L	System.Int64
uint64	64-bit unsigned integers	0UL, 19UL, 0x0800UL	System.UInt64
nativeint	Machine-sized signed integers	0n, 19n, 0x0800n	System.IntPtr
unativeint	Machine-sized unsigned integers	0un, 19un, 0x0800un	System.UIntPtr
single, float32	32-bit IEEE floating-point	0.0f, 19.7f, 1.3e4f	System.Single
double, float	64-bit IEEE floating-point	0.0, 19.7, 1.3e4	System.Double
decimal	High-precision decimal values	0M, 19M, 19.03M	System.Decimal
bigint	Arbitrarily large integers	0I, 19I	Math.BigInt
bignum	Arbitrary-precision rationals	0N, 19N	Math.BigNum
unit	The type with only one value	()	Core.Unit

جدول بالا کاملا واضح است و برنامه نویسان دات نت نظیر C# با انواع داده ای بالا آشنایی دارند. فقط در مورد گزینه آخر unit در فصل‌های بعدی توضیح خواهم داد.

Arithmetic Operators (عملگرهای محاسباتی)

Operator	Description	Sample Use on int	Sample Use on float
+	Unchecked addition	1 + 2	1.0 + 2.0
-	Unchecked subtraction	12 - 5	12.3 - 5.4
*	Unchecked multiplication	2 * 3	2.4 * 3.9
/	Division	5 / 2	5.0 / 2.0
%	Modulus	5 % 2	5.4 % 2.0
-	Unary negation	-(5+2)	-(5.4+2.4)

Simple String (کار با نوع داده رشته ای)

Example	Kind	Type
"Humpty Dumpty"	String	string
"c:\\Program Files"	String	string
@"c:\Program Files"	Verbatim string	string
"xyZy3d2"B	Literal byte array	byte []
'c'	Character	char

بعد از بررسی موارد بالا حالا به معرفی شناسه‌ها می‌پردازم. شناسه‌ها در F# راهی هستند برای اینکه شما به مقادیر نام اختصاص دهید. برای اختصاص نام به مقادیر کافیست از کلمه کلیدی let به همراه یک نام و علامت = و یک عبارت استفاده کنید. چیزی شبیه به تعریف متغیر در سایر زبان‌ها نظیر C#. دلیل اینکه در F# به جای واژه متغیر از شناسه استفاده می‌شود این است که شما می‌توانید به یک شناسه تابعی را نیز اختصاص دهید و مقدار شناسه‌ها دیگر قابل تغییر نیست. در F# کلمه متغیر یک واژه نادرست است چون زمانی که شما یه یک متغیر مقدار اختصاص می‌دهید، مقدار اون متغیر دیگه قایل تغییر نیست. برای همین اکثر برنامه نویسان F# به جای استفاده از واژه متغیر از واژه مقدار یا شناسه استفاه می‌کنند. برای همین از واژه متغیر برای نام گذاری استفاده نمی‌شود. (البته در F# در بعضی مواقع ما شناسه‌ها رو دوباره تعریف می‌کنیم که چیزی شبیه به استفاده از متغیر هاست ولی با اندکی تفاوت. در این فصل تمرکز ما بر روی شناسه هایی است که مقدارشان تغییر نمی‌کند ولی در فصل برنامه نویسی دستوری به تفصیل در این باره توضیح داده شده است)

```
let x = 42
```

در بالا یک شناسه به نام x تعریف شد که مقدار 42 را دریافت کرد. در #F یک شناسه می‌تواند دارای یک مقدار معین باشد یا به یک تابع اشاره کند. این بدین معنی است #F معنی حقیقی برای تابع و پارامترهای آن ندارد و همه چیز رو به عنوان مقدار در نظر می‌گیرد.

```
let myAdd = fun x y -> x + y
```

کد بالا تعریف یک شناسه به نام myAdd است که به تابعی اشاره می‌کنه که دو پارامتر ورودی دارد و در بدنه آن مقدار پارامترها با هم جمع می‌شوند.(تعریف توابع به صورت مفصل بحث خواهد شد.) نکته جالب این است که تابع تعریف شده نام ندارد و #F **دقیقا با توابع همون رفتاری رو داره که با شناسه‌ها دارد.**

```
let raisePowerTwo x = x ** 2.0
```

در کد بالا شناسه ای تعریف شده است با نام raisePowerTwo که یک پارامتر ورودی داره به نام x و در بدنه آن (هرچیزی که بعد از = قرار گیرد) مقدار x رو به توان دو می‌کنه.

نام گذاری شناسه ها

برای نام گذاری شناسه‌ها نام انتخابی یا باید با Underscore شروع شود یا با حروف. بعد از آن می‌تونید از اعداد هم استفاده کنید.(نظیر سایر زبان‌های برنامه نویسی)

#F از unicode هم پشتیبانی می‌کنه یعنی می‌تونید متغیری به صورت زیر رو تعریف کنید.

```
let مسعود = ""
```

اگر احساس می‌کنید که قوانین نام گذاری در #F کمی محدود کننده است می‌تونید از علامت '' '' استفاده کنید و در بین این علامت هر کاراکتری که می‌خواهید رو قرار دهید و #F اونو به عنوان نام شناسه قبول خواهد کرد. برای نمونه

```
let ``more? `` = true
```

یا

```
let ``class`` = "style"
```

حتی امکان استفاده از کلمات کلیدی هم نظیر class به این روش وجود دارد.

محدوده تعریف شناسه ها

به دلیل اینکه در #F از {} به عنوان شروع و اتمام محدوده استفاده نمی‌شود دونستن و شناختن محدوده توابع بسیار مهم و ضروری است. چون اگر از شناسه ای که در یک محدوده در دسترس نباشد استفاده کنید با خطای کامپایلر متوقف خواهید شد.

همون بحث متغیرهای محلی و سراسری (در سایر زبان ها) در این جا نیز صادق است یعنی در #F شناسه‌های سراسری و محلی خواهیم داشت. تمام شناسه ها، چه اون هایی که در توابع استفاده می‌شوند و چه اونهایی که به مقادیر اشاره می‌کنند محدودشون از نقطه ای که تعریف می‌شوند تا جایی که اتمام استفاده از اونهاست تعریف شده است. برای مثال اگر یک شناسه رو در بالای فایل تعریف کنید که یک مقدار دارد تا پایان SourceFile قابل استفاده است.(به دلیل نبود مفهوم کلاس از واژه sourceFile استفاده کردم). هم چنین شناسه هایی که در توابع تعریف می‌شوند فقط در همون توابع قابل استفاده هستند.

حالا سوال این است که با نبودن {} چگونه محدوده خود توابع مشخص میشود؟

در #F با استفاده از فضای خای یا space محدوده شناسه‌ها و توابع رو مشخص می‌کنیم. برای روشن شدن مطلب به مثال زیر دقت کنید.

```
let test a b =
    let dif = b - a
```

```
    let mid = dif / 2
    mid + a

printfn "(test 5 11) = %i" (test 5 11)
printfn "(test 11 5) = %i" (test 11 5)
```

ابتدا اختلاف بین دو ورودی محاسبه می‌شود و در یک شناسه به نام dif قرار می‌گیرد. برای اینکه مشخص شود که این شناسه خود عضو یک تابع دیگر به نام test است از 4 فضای خالی استفاده شده است. در خط بعدی شناسه mid مقدار شناسه dif رو بر 2 تقسیم می‌کند. در انتها نیز مقدار mid با مقدار a جمع می‌شود و حاصل برگشت داده می‌شود.(انتهای بدنه تابع)
نکته مهم : به جای استفاده از فضای خالی(space) نمی‌تونید از TAB استفاده کنید.

LIGHTWEIGHT SYNTAX یا VERBOSE SYNTAX

در #F دو نوع سبک کد نویسی وجود دارد. یکی lightweight و دیگری Verbose. البته اکثر برنامه نویسان از سبک lightweight که به صورت پیش فرض در #F تعبیه شده است استفاده می‌کنند ولی آشنایی با سبک verbose نیز به عنوان برنامه نویس #F ضروری است. ما نیز به تبعیت از سایرین از سبک lightweight استفاده خواهیم کرد ولی یک فصل به عنوان مطالب تکمیلی اختصاص دادم که تفاوت این دو سبک را در طی چندین مثال بیان میکند.
همان طور که قبلا بیان شد #F بر اساس زبان OCaml پیاده سازی شده است. زبان OCaml مانند #F، یک زبان LIGHTWEIGHT SYNTAX نیست. LIGHTWEIGHT SYNTAX بدین معنی است محدوده شناسه‌ها بر اساس فضای خالی بین اون‌ها مشخص می‌شود نه با ;. (البته استفاده از ; به صورت اختیاری است)
بازنویسی مثال بالا

```
let halfWay a b =
let dif = b - a in
let mid = dif / 2 in
mid + a
```

برای اینکه کامپایلر #F متوجه شود که قصد کدنویسی به سبک lightweight رو نداریم، باید در ابتدای هر فایل از دستور زیر استفاده کنیم.

```
#light "off"
```

عنوان: **توابع(Function)**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۷ ۱۴:۴۴
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

برنامه نویسی تابع گرا در یک جمله یعنی نوشتن توابع در پروژه و فراخوانی آن‌ها به همراه مقدار دهی به آرگومان‌های متناظر و دریافت خروجی در صورت نیاز. در F# پارامترهای یک تابع با پرانتز یا کاما از هم تمیز داده نمی‌شوند بلکه باید فقط از یک فضای خالی بین آن‌ها استفاده کنید.(البته می‌تونید برای خوانایی بهتر از پرانتز استفاده کنید)

```
let add x y = x + y
let result = add 4 5
printfn "(add 4 5) = %i" result
```

همان طور که می‌بینید تابعی به نام add داریم که دارای 2 پارامتر ورودی است به نام‌های x , y که فقط توسط یک فضای خالی از هم جدا شدند. حال به مثال دیگر توجه کنید.

```
let add x y = x + y
let result1 = add 4 5
let result2 = add 6 7
let finalResult = add result1 result2
```

در مثال بالا همان تابع add 2 بار فراخوانی شده است که یک بار مقدار خروجی تابع در یک شناسه به نام result1 و یک بار مقدار خروجی با مقادیر متفاوت در شناسه به نام result2 قرار گرفت. شناسه finalResult حاصل فراخوانی تابع add با مقادیر result1 , result2 است. می‌تونیم کد بالا رو به روش مناسب‌تری باز نویسی کنیم.

```
let add x y = x + y
let result =add (add 4 5) (add 6 7)
```

در اینجا برای خوانایی بهتر کد از پرانتز برای جداسازی مقدار پارامترها استفاده کردم.

خروجی توابع

کامپایلر F# آخرین مقداری که در تابع، تعریف و استفاده می‌شود را به عنوان مقدار بازگشتی و نوع آن را نوع بازگشتی می‌شناسد.

```
let cylinderVolume radius length : float =

   let pi = 3.14159
   length * pi * radius * radius
```

در مثال بالا خروجی تابع مقدار (length * pi * radius * radius) است و نو ع آن float می‌باشد.
یک مثال دیگر:

```
let sign num =
    if num > 0 then "positive"
    elif num < 0 then "negative"
    else "zero"
```

خروجی تابع بالا از نوع string است و مقدار آن با توجه به ورودی تابع positive یا negative یا zero خواهد بود.

تعریف پارامترهای تابع با ذکر نوع به صورت صریح

اگر هنگام تعریف توابع مایل باشید که نوع پارامترها را به صورت صریح تعیین کنید از روش زیر استفاده می‌کنیم.

```
let replace(str: string) =
```

```
    str.Replace("A", "a")
```

تعریف تابع به همراه دو پارامتر و ذکر نوع فقط برای یکی از پارامترها :

```
let addu1 (x : uint32) y =
    x + y
```

Pipe-Forward Operator

در #F روشی دیگری برای تعریف توابع وجود دارد که به pipe-Forward معروف است. فقط کافیست از اپراتور (<|) به صورت زیر استفاده کنید.

```
let (|>) x f = f x
```

کد بالا به این معنی است که تابعی یک پارامتر ورودی به نام x دارد و این پارامتر رو به تابع مورد نظر(هر تابعی که شما هنگام استفاده تعیین کنید) تحویل می‌دهد و خروجی را بر می‌گرداند. برای مثال

```
let result = 0.5 |> System.Math.Cos
```

یا

```
let add x y = x + y
let result = add 6 7 |> add 4 |> add 5
```

در مثال بالا ابتدا حاصل جمع 7 و 6 محاسبه می‌شود و نتیجه با 4 جمع می‌شود و دوباره نتیجه با 5 جمع می‌شود تا حاصل نهایی در result قرار گیرد. به نظر اکثر برنامه نویسان #F این روش نسبت به روش‌های قبلی خواناتر است. این روش همچنین مزایای دیگری نیز دارد که در مبحث Partial Functionها بحث خواهیم کرد.

Partial Fucntion Or Application

partial function به این معنی است که در هنگام فراخوانی یک تابع نیاز نیست که به تمام آرگومان‌های مورد نیاز مقدار اختصاص دهیم. برای نمونه در مثال بالا تابع add نیاز به 2 آرگومان ورودی داشت در حالی که فقط یک مقدار به آن پاس داده شد.

let result = add 6 7 |> add 4 دلیل برخورد #F با این مسئله این است که #F توابع رو به شکل مقدار در نظر می‌گیرد و اگر تمام مقادیر مورد نیاز یک تابع در هنگام فراخوانی تحویل داده نشود، از مقدار برگشت داده شده فراخوانی تابع قبلی استفاده خواهد کرد. البته این مورد همیشه خوشایند نیست. اما می‌تونیم با استفاده از پرانتز ر هنگام تعریف توابع مشخص کنیم که دقیقا نیاز به چند تا مقدار ورودی برای توابع داریم.

```
let sub (a, b) = a - b
let subFour = sub 4
```

کد بالا کامپایل نخواهد شد و خطای زیر رو مشاهده خواهید کرد.

```
prog.fs(15,19): error: FS0001: This expression has type
int
but is here used with type
'a * 'b
```

توابع بازگشتی

در مورد ماهیت توابع بازگشتی نیاز به توضیح نیست فقط در مورد نوع پیاده سازی اون در #F توضیح خواهم داد. برای تعریف

توابع به صورت بازگشتی کافیست از کلمه rec بعد از let استفاده کنیم(زمانی که قصد فراخوانی تابع رو در خود تابع داشته باشیم). مثال پایین به خوبی مسئله را روشن خواهد کرد.(پیاده سازی تابع فیبو ناچی)

```
let rec fib x =
match x with
| 1 -> 1
| 2 -> 1
| x -> fib (x - 1) + fib (x - 2)


printfn "(fib 2) = %i" (fib 2)
printfn "(fib 6) = %i" (fib 6)
printfn "(fib 11) = %i" (fib 11)
```

*درباره الگوی Matching در فصل بعد به صورت کامل توضیح خواهم داد.
خروجی برای مثال بالا به صورت خواهد شد.

```
(fib 2) = 1
(fib 6) = 8
(fib 11) = 89
```

توابع بازگشتی دو طرفه

گاهی اوقات توابع به صورت دوطرفه بازگشتی می‌شوند. یعنی فراخوانی توابع به صورت چرخشی انجام می‌شود. (فراخوانی یک تابع در تابع دیگر و بالعکس). به مثال زیر دقت کنید.

```
let rec Even x =
   if x = 0 then true
   else Odd (x - 1)
and Odd x =
   if x = 1 then true
   else Even (x - 1)
```

کاملا واضح است در تابع Even فراخوانی تابع Odd انجام می‌شود و در تابع Odd فراخوانی تابع Even. به این حالت mutual recursive می‌گویند.

ترکیب توابع

```
let firstFunction x = x + 1
let secondFunction x = x * 2
let newFunction = firstFunction >> secondFunction
let result = newFunction 100
```

در مثال بالا دو تابع به نام‌های firstFunction و secondFunction داریم. بااستفاده از (>>) دو تابع را با هم ترکیب می‌کنیم. خروجی بدین صورت محاسبه می‌شود که ابتدا تابع firstFucntion مقدار x را محاسبه می‌کند و حاصل به تابع secondFucntion پاس داده می‌شود. در نهایت یک تابع جدید به نام newFunction خواهیم داشت که مقدار نهایی محاسبه خواهد شد. خروجی مثال بالا 202 است.

توابع تودرتو

در F# امکان تعریف توابع تودرتو وجود دارد. بعنی می‌تونیم یک تابع را در یک تابع دیگر تعریف کنیم. فقط نکته مهم در امر استفاده از توابع به این شکل این است که توابع تودرتو فقط در همون تابعی که تعریف می‌شوند قایل استفاده هستند و محدوده این توابع در خود همون تابع است.

```
let sumOfDivisors n =
    let rec loop current max acc =

شروع تابع داخلی//

   if current > max then
            acc
```

```
        else
            if n % current = 0 then
                loop (current + 1) max (acc + current)
            else
                loop (current + 1) max acc

//ادامه بدنه تابع اصلی

 let start = 2
    let max = n / 2     (* largest factor, apart from n, cannot be > n / 2 *)
    let minSum = 1 + n  (* 1 and n are already factors of n *)
    loop start max minSum

printfn "%d" (sumOfDivisors 10)
```

در مثال بالا یک تابع تعریف کرده ایم به نام sumOfDivisors. در داخل این تابع یک تابع دیگر به نام loop داریم که از نوع بازگشتی است(به دلیل وجود rec بعد از let). بدنه تابع داخلی به صورت زیر است:

```
if current > max then
            acc
        else
            if n % current = 0 then
                loop (current + 1) max (acc + current)
            else
                loop (current + 1) max acc
```

خروجی مثال بالا برای ورودی 10 عدد 18 می‌باشد. مجموع مقصوم علیه‌های عدد 10 (1 + 2 + 5 + 10).

آیا می‌توان توابع را Overload کرد؟

در F# امکان overloading برای یک تابع وجود ندارد. ولی متدها را می‌توان overload کرد.(متدها در فصل شی گرایی توضیح داده می‌شود).

do keyword

زمانی که قصد اجرای یک کد را بدون تعریف یک تابع داشته باشیم باید از do استفاده کنیم. همچنین از do در انجام برخی عملیات پیش فرض در کلاس‌ها زیاد استفاده می‌کنیم.(در فصل شی گرایی با این مورد آشنا خواهید شد).

```
open System
open System.Windows.Forms

let form1 = new Form()
form1.Text <- "XYZ"

[<STAThread>]
do
   Application.Run(form1)
```

عنوان: **الگوی Matching**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۷ ۱۵:۳
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

الگوی Matching در واقع همون switch در اکثر زبان‌ها نظیر C# یا C++ است با این تفاوت که بسیار انعطاف پذیرتر و قدرتمندتر است. در برنامه نویسی تابع گرا، هدف اصلی از ایجاد توابع دریافت ورودی و اعمال برخی عملیات مورد نظر بر روی مقادیر با استفاده از تعریف حالات مختلف برای انتخاب عملیات است. الگوی Matching این امکان رو به ما می‌ده که با استفاده از حالات مختلف یک عملیات انتخاب شود و با توجه به ورودی یک سری دستورات رو اجرا کنه. ساختار کلی تعریف آن به شکل زیر است:

```
match expr with
| pat1 -> result1
| pat2 -> result2
| pat3 when expr2 -> result3
| _ -> defaultResult
```

راحت‌ترین روش استفاده از الگوی Matching هنگام کار با مقادیر است. اولین مثال رو هم در فصل قبل در بخش توابع بازگشتی با هم دیدیم.

```
let booleanToString x =
match x with false -> "False"
| _ -> "True"
```

در تابع بالا ورودی ما اگر false باشد "False" و اگر true باشد "True" برگشت داده می‌شود. _ در مثال بالا دقیقا همون default در switch سایر زبان هاست.

```
let stringToBoolean x =
match x with
| "True" | "true" -> true
| "False" | "false" -> false
| _ -> failwith "unexpected input"
```

در این مثال (دقیقا بر عکس مثال بالا) ابتدا یک string دریافت می‌شود اگر برابر "True" یا "true" بود مقدار true برگشت داده میشود و اگر برابر "False" یا "false" بود مقدار false برگشت داده می‌شود در غیر این صورت یک FailureException پرتاب می‌شود. خروجی مثال بالا در حالات مختلف به شکل زیر است:

```
printfn "(booleanToString true) = %s"
(booleanToString true)
printfn "(booleanToString false) = %s"
(booleanToString false)
printfn "(stringToBoolean \"True\") = %b"
(stringToBoolean "True")
printfn "(stringToBoolean \"false\") = %b"
(stringToBoolean "false")
```

خروجی : ("printfn "(stringToBoolean \"Hello\") = %b" (stringToBoolean "Hello

```
(booleanToString true) = True
(booleanToString false) = False
(stringToBoolean "True") = true
(stringToBoolean "false") = false
```

Microsoft.FSharp.Core.FailureException: unexpected input at FSI_0005.stringToBoolean(String x) at
<StartupCode$FSI_0005>.$FSI_0005.main@() هم چنین علاوه بر اینکه امکان استفاده از چند شناسه در این الگو وجود دارد، امکان استفاده از And , Or نیز در این الگو میسر است.

```
let myOr b1 b2 =
match b1, b2 with
| true, _ -> true  //b1 true , b2 true or false
| _, true -> true // b1 true or false , b2 true
| _ -> false


printfn "(myOr true false) = %b" (myOr true false)
printfn "(myOr false false) = %b" (myOr false false)
```

خروجی برای کدهای بالا به صورت زیر است:

```
(myOr true false) = true
(myOr false false) = false
```

استفاده از عبارت و شروط در الگوی Matching

در الگوی Matching اگر در بررسی ورودی الگو با یک مقدار نیاز شما را برطرف نمی‌کند استفاده از فیلترها و شروط مختلف هم مجاز است. برای مثال

```
let sign = function
    | 0 -> 0
    | x when x < 0 -> -1
    | x when x > 0 -> 1
```

مثال بالا برای تعیین علامت هر عدد ورودی به کار می‌رود. -1 برای عدد منفی و 1 برای عدد مثبت و 0 برای عدد 0.

عبارت if ... then ... else

استفاده از if در F# کاملا مشابه به استفاده از if در C# است و نیاز به توضیح ندارد. تنها تفاوت در if else است که در F# به صورت elif نوشته می‌شود.

ساختار کلی

```
if expr then
    expr
elif expr then
    expr
elif expr then
    expr
...
else
    expr
```

برای مثال الگوی Matching پایین رو به صورت if خواهیم نوشت.

```
let result =
match System.DateTime.Now.Second % 2 = 0 with
| true -> "heads"
| false -> "tails"
```

#با استفاده از if

```
let result =
if System.DateTime.Now.Second % 2 = 0 then
box "heads"
else
box false
printfn "%A" result
```

در پایان یک مثال مشترک رو به وسیله دستور swith case در C# و الگوی matching در F# پیاده سازی می‌کنیم.

C#	F#

switch (day) {
case 0: return "Sunday";
case 1: return "Monday";
case 2: return "Tuesday";
case 3: return "Wednesday";
case 4: return "Thursday";
case 5: return "Friday";
case 6: return "Saturday";
default:
throw new
ArgumentException("day");
}
	match day with
	| 0 -> "Sunday"
	| 1 -> "Monday"
	| 2 -> "Tuesday"
	| 3 -> "Wednesday"
	| 4 -> "Thursday"
	| 5 -> "Friday"
	| 6 -> "Saturday"
	| _ ->
	invalidArg "day" "Invalid day"

عنوان: **لیست ها و آرایه ها در F#**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۷ ۱۶:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

برای تعریف لیست در F# فقط کافیست از [] و برای جداسازی آیتم‌های موجود در لیست از عملگر :: (بخوانید cons) استفاده کنید. F# از لیست‌های خالی نیز پشتیبانی می‌کند. به مثال هایی از این دست توجه کنید

```
#1 let emptyList = []
#2 let oneItem = "one " :: []
#3 let twoItem = "one " :: "two " :: []
```

#1 تعریف یک لیست خالی
#2 تعریف یک لیست به همراه یک آیتم
#3 تعریف یک لیست به همراه دو آیتم
قبول دارم که دستورالعمل بالا برای مقدار دهی اولیه به لیست کمی طولانی و سخت است. برای همین می‌تونید از روش زیر هم استفاده کنید.

```
let shortHand = ["apples "; "pears"]
```

*کد بالا یک لیست با دو آیتم که از نوع رشته ای هستند تولید خواهد کرد.
می‌تونید از عملگر @ برای پیوستن دو لیست به هم نیز استفاده کنید.

```
let twoLists = ["one, "; "two, "] @ ["buckle "; "my "; "shoe "]
```

نکته : تمام آیتم‌های موجود در لیست باید از یک نوع باشند. یعنی امکان تعریف لیستی که دارای آیتم هایی با datatypeهای متفاوت باشد باعث تولید خطای کامپایلری می‌شود. اما اگر نیاز به لیستی دارید که باید چند datatype رو هم پوشش دهد می‌تونید از objectها استفاده کنید.

```
let objList = [box 1; box 2.0; box "three"]
```

در بالا یک لیست از objectها رو تعریف کرده ایم. فقط دقت کنید برای اینکه آیتم‌های موجود در لیست رو تبدیل به object کنیم از دستور box قبل از هر آیتم استفاده کردیم.

در هنگام استفاده از عملگرها @ و :: مقدار لیست تغییر نمی‌کند بلکه یک لیست جدید تولید خواهد شد.

```
#1 let one = ["one "]
#2 let two = "two " :: one
#3 let three = "three " :: two
#4 let rightWayRound = List.rev three

#5 let main() =
printfn "%A" one
printfn "%A" two
printfn "%A" three
printfn "%A" rightWayRound
```

#1 تعریف لیستی که دارای یک آیتم است.
#2 تعریف لیستی که دارای دو آیتم است(آیتم دوم لیست خود از نوع لیست است)
#3 تعریف لیستی که دارای سه آیتم است(ایتم دوم لیست خود از نوع لیستی است که دارای دو آیتم است)

از تابع List.rev برای معکوس کردن آیتم‌های لیست three استفاده کردیم و مقادیر در لیستی به نام rightWayRound قرار گرفت.

#5 تابع main برای چاپ اطلاعات لیست ها

بعد از اجرا خروجی زیر مشاهده می‌شود.

```
["one "]
["two "; "one "]
["three "; "two "; "one "]
["one "; "two "; "three "]
```

تفاوت بین لیست‌ها در #F و لیست و آرایه در دات نت(System.Collection.Generic)

	F#List	Net Array	Net List
#1 امکان تغییر در عناصر لیست	No	Yes	Yes
#2 امکان اضافه کردن عنصر جدید	No	No	Yes
#3 جستجو	On	O1	O1

#1 در #F بعد از ساختن یک لیست امکان تغییر در مقادیر عناصر آن وجود ندارد.

#2 در #F بعد از ساختن یک لیست دیگه نمی‌تونید یک عنصر جدید به لیست اضافه کنید.

#3 جستجوی در لیست‌های #F به نسبت لیست‌ها و آرایه‌های در دات نت کندتر عمل می‌کند.

استفاده از عبارات در لیست ها

برای تعریف محدوده در لیست می‌تونیم به راحتی از روش زیر استفاده کنیم

```
let rangeList = [1..99]
```

برای ساخت لیست‌ها به صورت داینامیک استفاده از حلقه‌های تکرار در لیست مجاز است.

```
let dynamicList = [for x in 1..99 -> x*x]
```

کد بالا معادل کد زیر در #C است.

```
for(int x=0;x<99 ; x++)
{
   myList.Add(x*x);
}
```

لیست‌ها و الگوی Matching

روش عادی برای کار با لیست‌ها در #F استفاده از الگوی Matching و توابع بازگشتی است.

```
let listOfList = [[2; 3; 5]; [7; 11; 13]; [17; 19; 23; 29]]

let rec concatList l =
match l with
| head :: tail -> head @ (concatList tail)
| [] -> []

let primes = concatList listOfList

printfn "%A" primes
```

در مثال بالا ابتدا یک لیست تعریف کردیم که دارای 3 آیتم است و هر آیتم آن خود یک لیست با سه آیتم است.(تمام آیتم‌ها از نوع داده عددی هستند). یک تابع بازگشتی برای پیمایش تمام آیتم‌های لیست نوشتم که در اون از الگوی Matching استفاده کردیم.
خروجی :

```
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

ماژول لیست

در جدول زیر تعدادی از توابع ماژول لیست رو مشاهده می‌کنید.

نام تابع	توضیحات
List.length	تابعی که طول لیست را برمی گرداند
List.head	تابعی برای برگشت عنصر اول لیست
List.tail	تمام عناصر لیست را بر میگرداند به جز عنصر اول
List.init	یک لیست با توجه به تعداد آیتم ایجاد می‌کند و یم تابع را بر روی تک تک عناصر لیست ایجاد می‌کند.
List.append	یک لیست را به عنوان ورودی دریافت می‌کند و به لیست مورد نظر اضافه می‌کند و مجموع دو لیست را برگشت می‌دهد
List.filter	فقط عناصری را برگشت می‌دهد که شرط مورد نظر بر روی آن‌ها مقدار true را برگشت دهد
List.map	یک تابع مورد نظر را بر روی تک تک عناصر لیست اجرا می‌کند و لیست جدید را برگشت می‌دهد
List.iter	یک تابع مورد نظر را بر روی تک تک عناصر لیست اجرا می‌کند
List.zip	مقادیر دو لیست را با هم تجمیع می‌کند و لیست جدید را برگشت می‌دهد. اگر طول 2 لیست ورودی یکی نباشد خطا رخ خواهدداد
List.unzip	درست برعکس تابع بالا عمل می‌کند
List.toArray	لیست را تبدیل به آرایه می‌کند
List.ofArray	آرایه را تبدیل به لیست می‌کند

مثال هایی از توابع بالا

```
List.head [5; 4; 3]
List.tail [5; 4; 3]
List.map (fun x -> x*x) [1; 2; 3]
List.filter (fun x -> x % 3 = 0) [2; 3; 5; 7; 9]
```

Sequence Collection seq در #F یک توالی از عناصری است که هم نوع باشند. عموما از sequenceها زمانی استفاده میکنیم که یک مجموعه از داده‌ها با تعداد زیاد و مرتب شده داشته باشیم ولی نیاز به استفاده از تمام عناصر آن نیست. کارایی sequence در مجموعه‌های با تعداد زیاد از listها به مراتب بهتر است. sequenceها را با تابع seq می‌شناسند که معادل IEnumerable در دات

نت است. بنابر این هر مجمو عه ای که IEnumerable رو در دات نت پیاده سازی کرده باشد در #F با seq قابل استفاده است.

مثال هایی از نحوه استفاده seq

#1 seq بامحدوده 1 تا 100 و توالی 10

```
seq { 0 .. 10 .. 100 }
```

#2 استفاده از حلقه‌های تکرار برای تعریف محدوده و توالی در seq

```
seq { for i in 1 .. 10 do yield i * i }
```

#3 استفاده از <- به جای yield

```
seq { for i in 1 .. 10 -> i * i }
```

#4 استفاده از حلقه for به همراه شرط برای فیلتر کردن

```
let isprime n =
    let rec check i =
        i > n/2 || (n % i <> 0 && check (i + 1))
    check 2

let aSequence = seq { for n in 1..100 do if isprime n then yield n }
```

چگونگی استفاده از توابع seq

در این بخش به ارائه مثال هایی کاربردی‌تر از چگونگی استفاده از seq در #F می‌پردازیم. برای شروع نحوه ساخت یک seq خالی یا empty رو خواهم گفت.

```
let seqEmpty = Seq.empty
```

روش ساخت یک seq که فقط یک عنصر را برگشت می‌دهد.

```
let seqOne = Seq.singleton 10
```

برای ساختن یک seq همانند لیست‌ها می‌تونیم از seq.init استفاده کنیم. عدد 5 که بلافاصله بعد از تابع seq.init آمده است نشان دهنده تعداد آیتم‌ها موجود در seq خواهد بود. seq.iter هم یک تابع مورد نظر رو بر روی تک تک عناصر seq اجرا خواهد کرد.(همانند list.iter)

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

خروجی مثال بالا

```
0 10 20 30 40
```

با استفاده از توابع seq.ofList , seq.ofArray می‌تونیم seq مورد نظر خود را از لیست یا آرایه مورد نظر بسازیم.

```
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

البته این نکته رو هم یادآور بشم که به کمک عملیات تبدیل نوع(type casting) هم می‌تونیم آرایه رو به seq تبدیل کنیم. به صورت زیر

```
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>
```

برای مشخص کردن اینکه آیا یک آیتم در seq موجود است یا نه می‌تونیم از seq.exists به صورت زیر استفاده کنیم.

```
let containsNumber number seq1 = Seq.exists (fun elem -> elem = number) seq1
let seq0to3 = seq {0 .. 3}
printfn "For sequence %A, contains zero is %b" seq0to3 (containsNumber 0 seq0to3)
```

اگر seq پاس داده شده به تابع exists خالی باشد یا یک ArgumentNullException متوقف خواهید شد.

برای جستجو و پیدا کردن یک آیتم در seq می‌تونیم از seq.find استفاده کنیم.

```
let isDivisibleBy number elem = elem % number = 0
let result = Seq.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

دقت کنید که اگر هیچ آیتمی در sequence با predicate مورد نظر پیدا نشود یک KeyNotFoundException رخ خواهد داد. در صورتی که مایل نباشید که استثنا رخ دهد می‌توانید از تابع seq.tryFind استفاده کنید. هم چنین خالی بودن sequence ورودی باعث ArgumentNullExceptionخواهد شد.

استفاده از lambda expression در توابع

lamdaExpressoion از توانایی‌ها مورد علاقه برنامه نویسان دات نت است و کمتر کسی است حاضر به استفاده از آن در کوئری‌های linq نباشد. در #F نیز می‌توانید از lambda Expression استفاده کنید. در ادامه به بررسی مثال هایی از این دست خواهیم پرداخت.

تابع skipWhile

همانند skipWhile در linq عمل می‌کند. یعنی یک predicate مورد نظر را بر روی تک تک عناصر یک لیست اجرا می‌کند و آیتم هایی که شرط برای آن‌ها true باشد نادیده گرفته میشوند و مابقی آیتم‌ها برگشت داده می‌شوند.

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let mySeqSkipWhileLessThan10 = Seq.skipWhile (
```

fun elem -> elem < 10

```
) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

می‌بینید که predicate مورد نظر برای تابع skipWhile به صورت lambda expression است که با رنگ متفاوت نمایش داده شده است.(استفاده از کلمه fun).

خروجی به صورت زیر است:

```
16 25 36 49 64 81 100
```

برای بازگرداندن یک تعداد مشخص از آیتم‌های seq می‌تونید از توابع seq.take یا seq.truncate استفاده کنید. ابتدا باید تعداد مورد نظر و بعد لیست مورد نظر را به عنوان پارامتر مقدار دهی کنید.

مثال:

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""


#1 truncatedSeq |> printSeq
#3 takenSeq |> printSeq
```

خروجی

```
1 4 9 16 25 //truncate
1 4 9 16 25 //take
```

Tuples

tuples در #F به گروهی از مقادیر بی نام ولی مرتب شده که می‌توانند انواع متفاوت هم داشته باشند گفته می‌شود. ساختار کلی آن به صورت (element , ... , element) است که هر element خود می‌تواند یک عبارت نیز باشد.(مشابه کلاس Tuple در #C که به صورت generic استفاده می‌کنیم)

```
// Tuple of two integers.
( 1, 2 )

// Triple of strings.
( "one", "two", "three" )

// Tuple of unknown types.
( a, b )

// Tuple that has mixed types.
( "one", 1, 2.0 )

// Tuple of integer expressions.
( a + 1, b + 1)
```

نکات استفاده از tuple

#1 می‌تونیم از الگوی Matching برای دسترسی به عناصر tuple استفاده کنیم.

```
let print tuple1 =
   match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

#2 میتونیم از let برای تعربف الگوی tuple استفاده کنیم.

```
let (a, b) = (1, 2)
```

#3 توابع fst و snd مقادیر اول و دوم هر tuple رو بازگشت می‌دهند

```
let c = fst (1, 2) // return 1
let d = snd (1, 2)// return 2
```

#4 تابعی برای بازگشت عنصر سوم یک tuple وجود ندارد ولی این تابع رو با هم می‌نویسیم:

```
let third (_, _, c) = c
```

کاربرد tuple در کجاست

زمانی که یک تابع باید بیش از یک مقدار را بازگشت دهد از tupleها استفاده می‌کنیم. برای مثال

```
let divRem a b =
   let x = a / b
   let y = a % b
   (x, y)
```

خروجی تابع divRem از نوع tuple که دارای 2 مقدار است می‌باشد.

عنوان: **خطا ها و مدیریت خطا (Exception Handling)**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۸ ۰:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

مدیریت خطا در F# شبیه به الگوی try catch finally در C# است. برای تعریف خطا از کلمه کلیدی exception استفاده می‌کنیم و یک نام رو به اون اختصاص می‌دهیم و می‌تونیم به صورت اختیاری یک نوع داده رو هم برای این خطا با استفاده از کلمه کلیدی of تعیین کنیم.

```
exception myError of int
```

با استفاده از دستور raise می‌تونیم یک exception رو پرتاب کنیم.(به دلیل اینکه در دات نت از دستور throw به معنی پرتاب کردن استفاده می‌کنیم این جا نیز از همین لغت استفاده کردم کما اینکه در F# دستور raise جایگزین throw شده است). البته در جاهایی که قصد ما از پرتاب exception فقط متوقف کردن عملیات و نمایش یک خطا است می‌تونیم از دستور failwith به همراه یک پیغام نیز استفاده کنیم.(یک نمونه از آن را در فصل‌های قبلی مشاهده کردید)
ساختار کلی try catch finally در F# به صورت زیر است.(تنها تفاوت در کلمه with به جای catch است)

```
try
// try code here
with
//catch statement here
```

یا به صورت

```
try
// try code here
finally
//finally statement here
```

*نکته مهم: در F# شما اجازه استفاده از finally رو به همراه with ندارید.به همین دلیل من این ساختارو به دو صورت بالا نوشتم.

یک مثال از try with:

```
exception WrongSecond of int//یک exception تعریف می‌کنیم

let primes =
[ 2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59 ]

// وجود دارد یا نه prime یک تابع برای تست اینکه آیا ثانیه الان در لیست
let testSecond() =
try
let currentSecond = System.DateTime.Now.Second in

// شرط برای اینکه مشخص شود که ثانیه در لیست است یا خیر
if List.exists (fun x -> x = currentSecond) primes then

// اگر بود یک خطا تولید می‌شود
failwith "A prime second"

else

// پرتاب میشود wrongSecond اگر نبود یک استثنا از نوع
raise (WrongSecond currentSecond)

with
// catch کردن استثناها

WrongSecond x ->
printf "The current was %i, which is not prime" x
```

در کد با در هر خط توضیحات لازم داده شده است. نکته قابل ذکر این است که در #C زمانی که قصد داشته باشیم یک استثنا جدید ایجاد کنیم باید کلاسی جدیدی که از کلاس System.Exception ارث برده باشد(یا هر کلاس دیگری که خود از این System.Exception ارث برده است) ایجاد کنیم و کدهای مورد نظر رو در اون قرار بدیم. ولی در اینجا (در قسمتی که رنگ آن متفاوت است) به راحتی توانستیم یک استثنا جدید بر اساس نیاز بسازیم.

یک مثال از try finally :

```
// تابعی برای نوشتن فایل
let writeToFile() =
//ابتدا فایل به صورت متنی ساخته می‌شود
let file = System.IO.File.CreateText("test.txt")
try
// متن مورد نظر در فایل نوشته می‌شود
file.WriteLine("Hello F# users")
finally
//فایل مورد نظر بسته می‌شود.این دستور حتی اگر در هنگام نوشتن فایل استثنا هم رخ بدهد اجرا خواهد شد
file.Dispose()
```

عملکرد finally در #F دقیقا مشابه با عملکرد finally در #C است. یعنی دستورات بلوک finally همواره (چه استثنا رخ بدهد و چه رخ ندهد) اجرا خواهد شد.

***توجه :** برنامه نویسانی که قبلا با OCaml کدنویسی کرده اند هنگام برنامه نویسی #F از raise کردن‌های زیاد و بی مورد استثناها خودداری کنند. به دلیل نوع معماری CLR پرتاب کردن استثنا و مدیریت آن کمی هزینه بر است (بیشتر از زبان Ocaml). البته این مسئله در زبان‌های تحت دات نت نیز مطرح است کما اینکه در #C نیز مدیریت استثناها رو در بالاترین لایه انجام می‌دهیم و از catch کردن بی مورد استثنائات در لایه‌های زیرین خودداری می‌کنیم.

یک مثال از الگوی Matching در try with

```
let getNumber msg =
    printf msg;
    try
        int32(System.Console.ReadLine())
    with
        | :? System.FormatException -> -1
        | :? System.OverflowException -> System.Int32.MinValue
        | :? System.ArgumentNullException -> 0
```

عنوان: **نکاتی درباره برنامه نویسی دستوری(امری)**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۹ ۰:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

در این فصل نکاتی را درباره برنامه نویسی دستوری در F# فرا خواهیم گرفت. برای شروع از mutale خواهیم گفت.

mutable Keyword

در فصل دوم(شناسه ها) گفته شد که برای یک شناسه امکان تغییر مقدار وجود ندارد. اما در F# راهی وجود دارد که در صورت نیاز بتوانیم مقدار یک شناسه را تغییر دهیم.در F# هرگاه بخواهیم شناسه ای تعریف کنیم که بتوان در هر زمان مقدار شناسه رو به دلخواه تغییر داد از کلمه کلیدی mutable کمک می‌گیریم و برای تغییر مقادیر شناسه‌ها کافیست از علامت (<-) استفاده کنیم. به یک مثال در این زمینه دقت کنید:

```
#1 let mutable phrase = "Can it change? "

#2 printfn "%s" phrase

#3 phrase <- "yes, it can."

#4 printfn "%s" phrase
```

در خط اول یک شناسه را به صورت mutable(تغییر پذیر) تعریف کردیم و در خط سوم با استفاده از (<-) مقدار شناسه رو update کردیم. خروجی مثال بالا به صورت زیر است:

```
Can it change?
yes, it can.
```

نکته اول : در این روش هنگام update کردن مقدار شناسه حتما باید مقدار جدید از نوع مقدار قبلی باشد در غیر این صورت با خطای کامپایلری متوقف خواهید شد.

```
#1 let mutable phrase = "Can it change?  "

#3 phrase <- 1
```

اجرای کد بالا خطای زیر را به همراه خواهد داشت.(خطا کاملا واضح است و نیاز به توضیح دیده نمی‌شود)

```
Prog.fs(9,10): error: FS0001: This expression has type
int
but is here used with type
string
```

نکته دوم :ابتدا به مثال زیر توجه کنید.

```
let redefineX() =
let x = "One"
printfn "Redefining:\r\nx = %s" x
if true then
   let x = "Two"
printfn "x = %s" x
printfn "x = %s" x
```

در مثال بالا در تابع redefineX یک شناسه به نام x تعریف کردم با مقدار "One". یک بار مقدار شناسه x رو چاپ می‌کنیم و بعد دوباره بعد از شرط true یک شناسه دیگر با همون نام یعنی x تعریف شده است و در انتها هم دو دستور چاپ. ابتدا خروجی مثال بالا رو با هم مشاهده می‌کنیم.

```
Redefining:
x = One
x = Two
x = One
```

همان طور که میبینید شناسه دوم x بعد از تعریف دارای مقدار جدید Two بود و بعد از اتمام محدوده(scope) مقدار x دوباره به One تغییر کرد.(بهتر است بگوییم منظور از دستور print x سوم اشاره به شناسه x اول برنامه است). این رفتار مورد انتظار ما در هنگام استفاده از روش تعریف مجدد شناسه هاست. حال به بررسی رفتار muatable در این حالت می‌پردازیم.

```
let mutableX() =
let mutable x = "One"
printfn "Mutating:\r\nx = %s" x
if true then
   x <- "Two"
printfn "x = %s" x
printfn "x = %s" x
```

تنها تفاوت در استفاده از mutable keyword و (<-) است. خروجی مثال بالا نیز به صورت زیر خواهد بود. کاملا واضح است که مقدار شناسه x بعد از تغییر و اتمام محدوده(scope) هم چنان Two خواهد بود.

```
Mutating:
x = One
x = Two
x = Two
```

Reference Cells

روشی برای استفاده از شناسه‌ها به صورت mutable است. با این روش می‌تونید شناسه هایی تعریف کنید که امکان تغییر مقدار برای اون‌ها وجود دارد. زمانی که از این روش برای مقدار دهی به شناسه‌ها استفاده کنیم یک کپی از مقدار مورد نظر به شناسه اختصاص داده می‌شود نه آدرس مقدار در حافظه.
به جدول زیر توجه کنید:

Definition	Description	Member Or Field
let (!) r = r.contents	مقدار مشخص شده را برگشت می‌دهد	(derefence operator)!
let (:=) r x = r.contents <- x	مقدار مشخص شده را تغییر می‌دهد	(Assignment operator)=:
let ref x = { contents = x }	یک مقدار را در یک reference cell جدید کپسوله می‌کند	ref operator
member x.Value = x.contents	برای عملیات get یا set مقدار مشخص شده	Value Property
let ref x = { contents = x }	برای عملیات get یا set مقدار مشخص شده	contents record field

یک مثال:

```
let refVar = ref 6

refVar := 50



printfn "%d" !refVar
```

خروجی مثال بالا 50 خواهد بود.

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

خروجی مثال بالا:

```
10
10
11
12
```

خصیصه اختیاری در F#

در F# زمانی از خصیصه اختیاری استفاده می‌کنیم که برای یک متغیر مقدار وجود نداشته باشد. option در F# نوعی است که می‌تواند هم مقدار داشته باشد و هم نداشته باشد.

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

از None زمانی استفاده می‌کنیم که option مقدار نداشته باشد و از Some زمانی استفاده می‌کنیم که option مقدار داشته باشد.

```
let exists (x : int option) =
    match x with
    | Some(x) -> true
    | None -> false
```

در مثال بالا ورودی تابع exists از نوع int و به صورت اختیاری تعریف شده است.(معادل با ?int یا<int>Nullable در C#) در صورتی که x مقدار داشته باشد مقدار true در غیر این صورت مقدار false را برگشت می‌دهد.

چگونگی استفاده از option

مثال

```
let rec tryFindMatch pred list =
    match list with
    | head :: tail -> if pred(head)
                      then Some(head)
                      else tryFindMatch pred tail
    | [] -> None

let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ] //برابر با 100 است

let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]// برابر با None است
```

یک تابع به نام tryFindMatch داریم با دو پارامتر ورودی. با استفاده از الگوی Matching از عنصر ابتدا تا انتها را در لیست (پارامتر

list) با مقدار پارامتر pred مقایسه می‌کنیم. اگر مقادیر برابر بودند مقدار head در غیر این صورت None(یعنی option مقدار ندارد) برگشت داده می‌شود.
یک مثال کاربردی تر

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
                None
```

در مثال بالا از optionها برای بررسی وجود یا عدم وجود فایل‌های فیزیکی استفاده کردم.

Enumeration

تقریبا همه با نوع داده شمارشی یا enums آشنایی دارند. در اینجا فقط به نحوه پیاده سازی آن در F# می‌پردازیم. ساختار کلی تعریف آن به صورت زیر است:

```
type enum-name =
   | value1 = integer-literal1
   | value2 = integer-literal2
  ...
```

یک مثال از تعریف:

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2
```

نحوه استفاده

```
let col1 : Color = Color.Red
```

enums فقط از انواع داده ای sbyte, byte, int16, uint16, int32, uint32, int64, uint16, uint64, char پشتیبانی می‌کند که البته مقدار پیش فرض آن Int32 است. در صورتی که بخواهیم صریحا نوع داده ای را ذکر کنیم به صورت زیر عمل می‌شود.

```
type uColor =
    | Red = 0u
    | Green = 1u
    | Blue = 2u
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

توضیح درباره use

در دات نت خیلی از اشیا هستند که اینترفیس IDisposable رو پیاده سازی کرده اند. این بدین معنی است که حتما یک متد به نام dispose برای این اشیا وجود دارد که فراخوانی آن به طور قطع باعث بازگرداندن حافظه ای که در اختیار این کلاس‌ها بود می‌شود. برای راحتی کار در C# یک عبارت به نام using وجود دارد که در انتها بلاک متد dispose شی مربوطه را فراخوانی می‌کند.

```
using(var writer = new StreamWriter(filePath))
{

}
```

در F# نیز امکان استفاده از این عبارت با اندکی تفاوت وجود دارد.مثال:

```
let writeToFile fileName =
    use sw = new System.IO.StreamWriter(fileName : string)
    sw.Write("Hello ")
```

Units Of Measure

در #F اعداد دارای علامت و اعداد شناور دارای وابستگی با واحدهای اندازه گیری هستند که به نوعی معرف اندازه و حجم و مقدار و ... هستند. در #F شما مجاز به تعریف واحدهای اندازه گیری خاص خود هستید و در این تعاریف نوع عملیات اندازه گیری را مشخص می‌کنید. مزیت اصلی استفاده از این روش جلوگیری از رخ دادن خطاهای کامپایلر در پروژه است. ساختار کلی تعریف:

```
[<Measure>] type unit-name [ = measure ]
```

یک مثال از تعریف واحد cm:

```
[<Measure>] type cm
```

مثالی از تعریف میلی لیتر:

```
[<Measure>] type ml = cm^3
```

برای استفاده از این واحدها می‌تونید به روش زیر عمل کنید.

```
let value = 1.0<cm>
```

توابع تبدیل واحد ها

قدرت اصلی واحدهای اندازه گیری #F در توابع تبدیل است. تعریف توابع تبدیل به صورت زیر می‌باشد:

```
[<Measure>] type g                  تعریف واحد گرم
[<Measure>] type kg                 تعریف واحد کیلوگرم
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>    تعریف تابع تبدیل
```

یک مثال دیگر :

```
[<Measure>] type degC // دما بر حسب سلسیوس
[<Measure>] type degF // دما بر حسب فارنهایت

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF> // تابع تبدیل
سلسیوس به فارنهایت
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>) // تابع تبدیل
فارنهایت به سلسیوس

let degreesFahrenheit temp = temp * 1.0<degF> // درجه به فارنهایت
let degreesCelsius temp = temp * 1.0<degC> // درجه به سلسیوس

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let mutable floatValue = 0.
if System.Double.TryParse(input, &floatValue)// اگر ورودی عدد بود
   then
      printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
   else
      printfn "Error parsing input."
```

خروجی مثال بالا :

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

عنوان: **حلقه های تکرار**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۹ ۱:۲۳
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

در #F سه نوع حلقه تکرار وجود دارد. مفهوم حلقه‌های تکرار در #F مانند سایر زبان‌های برنامه نویسی است. در این جا فقط به syntax و نوع کد نویسی اشاره خواهیم داشت.

انواع حلقه‌های تکرار

حلقه تکرار for in

حلقه تکرار for to

حلقه تکرار while do

در ادامه به بررسی و پیاده سازی مثال برای هر سه حلقه می‌پردازیم

#1 حلقه for ساده

```
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
   printfn "%d" i
```

خروجی:

```
1
5
100
450
788
```

#2حلقه for با تعداد پرش دو(1 تا 10 با تعداد پرش 2)

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

خروجی

```
1 3 5 7 9
```

#3 حلقه for با استفاده از محدوده کاراکتر ها

```
for c in 'a' .. 'z' do
    printf "%c " c
```

خروجی

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

#4 حلقه for به صورت شمارش معکوس

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

خروجی :

```
10 9 8 7 6 5 4 3 2 1
```

5#حلقه for که شروع و اتمام محدوده آن به صورت عبارت است.

```
let beginning x y = x - 2*y
let ending x y = x + 2*y

  for i in (beginning x y) .. (ending x y) do
     printf "%d " i
```

خروجی مثال بالا با ورودی‌های 10 و 4 به صورت زیر خواهد بود

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

6# حلقه for to

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

خروجی

```
1 2 3 4 5 6 7 8 9 10
```

7# حلقه for to به صورت شمارش معکوس

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

خروجی

```
10 9 8 7 6 5 4 3 2 1
```

8# حلقه for to با استفاده از محدوده شروع و اتمام به صورت عبارت

```
for i = (beginning x y) to (ending x y) do
     printf "%d " i
```

خروجی مثال بالا برای ورودی‌های 10 و 4 به صورت زیر است

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

9# حلقه while do

ساختار کلی آن به صورت زیر است.

```
while test-expression do
   body-expression
```

10#مثال کامل از حلقه while do

```
open System

let main() =
    let password = "monkey"//شناسه برای مقدار رمز عبور
    let mutable guess = String.Empty// شناسه برای حدس رمز عبور
    let mutable attempts = 0//تعداد دفعات تست

    while password <> guess && attempts < 3 do// تا زمانی که رمز عبور با حدس آن برابر نیست و تعداد دفعات تکرار کمتر از سه است
        Console.Write("What's the password? ")//چاپ پیغام در خروجی
        attempts <- attempts + 1//مقدار دفعات یکی افزایش می‌یابد
        guess <- Console.ReadLine()//حدس رمز عبور از ورودی دریافت می‌شود

    if password = guess then// اگر رمز عبور با حدس آن یکی بود
        Console.WriteLine("You got the password right!")// پیغام موفقیت
    else
        Console.WriteLine("You didn't guess the password")//پیغام عدم موفقیت

    Console.ReadKey(true) |> ignore//منتظر ورودی برای خروج از برنامه
```

تنها نکته قابل ذکر در مثال بالا استفاده از mutable keyword برای تعریف شناسه attempts است. (طبق توضیحات فصل قبل) به دلیل اینکه تعداد دفعات تکرار برای ما مهم است و نمی‌خواهیم در هر محدوده این مقدار به حالت قبلی خود بازگردد از mutable استفاده کردیم.

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)
```

عنوان: **ماژول ها و فضای نام(namespace)**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۲۱ ۱:۳۶
آدرس: www.dotnettips.info
گروه‌ها: F#, Programming

در #F ماژول به گروهی از کدها، توابع، انواع داده‌ها و شناسه‌ها گفته می‌شود و کاربرد اصلی آن برای قرارگیری کدها مرتبط به هم در یک فایل است و هم چنین از تناقض نام‌ها جلوگیری می‌کند. در #F در صورتی که توسط برنامه نویس ماژول تعریف نشود هر source file یک ماژول در نظر گرفته می‌شود. برای مثال:

```
// In the file program.fs.
let x = 40
```

بعد از کامپایل تبدیل به کد زیر می‌شود.

```
module Program
let x = 40
```

هم چنین امکان تعریف چند ماژول در یک source file نیز میسر است. به این صورت که باید برای هر ماژول محلی یک نام اختصاص دهید. در مثال بعدی دو تا ماژول را در یک فایل به نام mySourceFile قرار می‌دهیم.

```
module MyModule1 =
     let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

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

در آخرین خط همان طور که مشاهده می‌کنید با استفاده از نام ماژول می‌توانیم به تعاریف موجود در ماژول دسترسی داشته باشیم.(MyModule1.module1Function).

استفاده از یک ماژول در فایل‌های دیگر.
گاهی اوقات نیاز به استفاده از تعاریف و توابع موجود در ماژولی داریم که در یک فایل دیگر قرار دارد. در این حالت باید به روش زیر عمل کنیم.
فرض بر این است ماژول زیر در یک فایل به نام ArithmeticFile قرار دارد.

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

حال قصد استفاده از توابع بالا رو در یک فایل و ماژول دیگر داریم.
#1 روش اول (دقیقا مشابه روش قبل از نام ماژول استفاه می‌کنیم)

```
let result1 = Arithmetic.add 5 9
```

#2 روش دوم(استفاده از open)

```
open Arithmetic
let result2 = add 5 9
```

ماژول‌های تودرتو

در #F می‌توانیم یک ماژول را درون ماژول دیگر تعریف کنیم یا به عبارت دیگر می‌توانیم ماژولی داشته باشیم که خود شامل چند تا ماژول دیگر باشد. مانند:

```
module Y =
    let x = 1

    module Z =
        let z = 5
```

روش تعریف ماژول‌های تودرتو در #F در نگاه اول کمی عجیب به نظر میرسه. جداسازی ماژول‌های تودرتو به وسیله دندانه گذاری یا تورفتگی انجام می‌شود. ماژول Z در مثال بالا به اندازه چهار فضای خالی جلوتر نسبت به ماژول Y قرار دارد در نتیجه به عنوان ماژول داخلی Y معرفی می‌شود.

```
module Y =
    let x = 1

module Z =
    let z = 5
```

در مثال بالا به دلیل اینکه ماژول Z و Y از نظر فضای خالی در یک ردیف قرار دارند در نتیجه ماژول تودرتو نیستند. حال به مثال بعدی توجه کنید.

```
module Y =
module Z =
    let z = 5
```

در این مثال ماژول X به عنوان ماژول داخلی Y حساب می‌شود. دلیلش هم این است که ماژول Y بدنه ندارد درنتیجه مازول Z بلافاصله بعد از آن قرار میگیرد که کامپایلر اونو به عنوان مازول داخلی حساب می‌کنه. اما برای اینکه مطمئن شود که قصد شما تولید ماژول تودرتو بود یک Warning میدهد. برای اینکه Warningv رو مشاهده نکنیم می تونیم کد بالا رو به صورت زیر بازنویسی کنیم:

```
module Y =
    module Z =
        let z = 5
```

فضای نام (namespace)

مفهوم فضای نام کاملا مشابه مفهوم فضای نام در #C است و راهی است برای کپسوله سازی کدها در برنامه. مفهوم namespace با مفهوم module کمی متفاوت است.

ساختار کلی

```
namespace [parent-namespaces.]identifier
```

چند نکته درباره namespace

1# اگر قصد داشته باشید که از فضای نام در کدهای خود استفاده کنید باید اولین تعریف در source file برنامه تعریف namespace باشد.

2# امکان تعریف شناسه یا تابع به صورت مستقیم در namespace وجود ندارد بلکه این تعاریف باید در ماژول‌ها یا typeها نظیر تعریف کلاس قرار گیرند.

3# امکان تعریف فضای نام با استفاده از تعاریف ماژول نیز وجود دارد(در ادامه به بررسی یک مثال در این زمینه می‌پردازیم)

تعریف namespace به صورت مستقیم:

```
namespace Model

type Car =
    member this.Name = "BMW"

module SetCarName =
    let CarName = "Pride"
```

تعریف namepsace به صورت غیر مستقیم (استفاده از module)

```
module Model.Car

module SetCarName =
    let CarName = "Pride"
```

فضای نام‌های تودرتو

همانند ماژول‌ها امکان تعریف فضای نام تودرتو نیز وجود دارد. یک مثال در این زمینه:

```
namespace Outer

    type OuterMyClass() =
       member this.X(x) = x + 1

namespace Outer.Inner

    type InnerMyClass() =
       member this.Prop1 = "X"
```

همانند فضای نام‌های در C# با استفاده از (.) می‌توانیم فضای نام‌های تودرتو ایجاد کنیم. در مثال بالا فضای نام Inner به عنوان فضای نام داخلی Outer تعریف شد است. برای دسترسی به کلاس InnerMyClass باید تمام مسیر فضای نام رو ذکر کنیم.

```
Outer.Inner.InnerMyClass
```

عنوان: **مقایسه بین سبک کدنویسی Lightweight و Verbose**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۲۲ ۲:۱۰
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

در این فصل با ذکر مثال،به مقایسه نحوه کدنویسی در F# با استفاده از دو نوع سبک Lightweight و Verbose می‌پردازیم.

استفاده از expression ها

```
//Lightweight



<expression1>
<expression2>

//Verbose
<expression1>; <expression2>
```

استفاده از letهای تودرتو

```
//Lightweight

let f x =
    let a = 1
    let b = 2
    x + a + b

//Verbose

let f x =
    let a = 1 in
    let b = 2 in
    x + a + b
```

محدوده کد

```
//Lightweight



<expression1>
    <expression2>
    ...

//Verbose

    begin
        <expression1>;
        <expression2>;
    end
```

حلقه تکرار for do

```
//Lightweight

for counter = start to finish do
    ...


 //Verbose


 for counter = start .. finish do
    ...
    done
```

حلقه تکرار while do

```
//Lightweight


 while <condition> do
    ...

 //Verbose


 while <condition> do
    ...
    done
```

حلقه تکرار for in

```
//Lightweight


for var in start .. finish do
    ...

 //Verbose

 for var in start .. finish do
    ...
    done
```

دستور do

```
//Lightweight
do ...

 //Verbose
 do ... in
```

تعریف record

```
//Lightweight


type <record-name> =
    {
        <field-declarations>
    }
    <value-or-member-definitions>

 //Verbose

 type <record-name> =
    {
        <field-declarations>
    }
    with
        <value-or-member-definitions>
    end
```

تعریف class

```
//Lightweight


type <class-name>(<params>) =
    ...

//Verbose



 type <class-name>(<params>) =
    class
        ...
    end
```

تعریف structure

```
//Lightweight
```

```
[<StructAttribute>]
type <structure-name> =
    ...
```

```
//Verbose

 type <structure-name> =
    struct
        ...
    end
```

تعریف Interface

```
//Lightweight

type <interface-name> =
    ...



//Verbose


 type <interface-name> =
    interface
        ...
    end
```

پیاده سازی Interface

```
//Lightweight

interface <interface-name>
    with
        <value-or-member-definitions>


 //Verbose


 interface <interface-name>
    with
        <value-or-member-definitions>
    end
```

تعریف module

```
//Lightweight

module <module-name> =
```

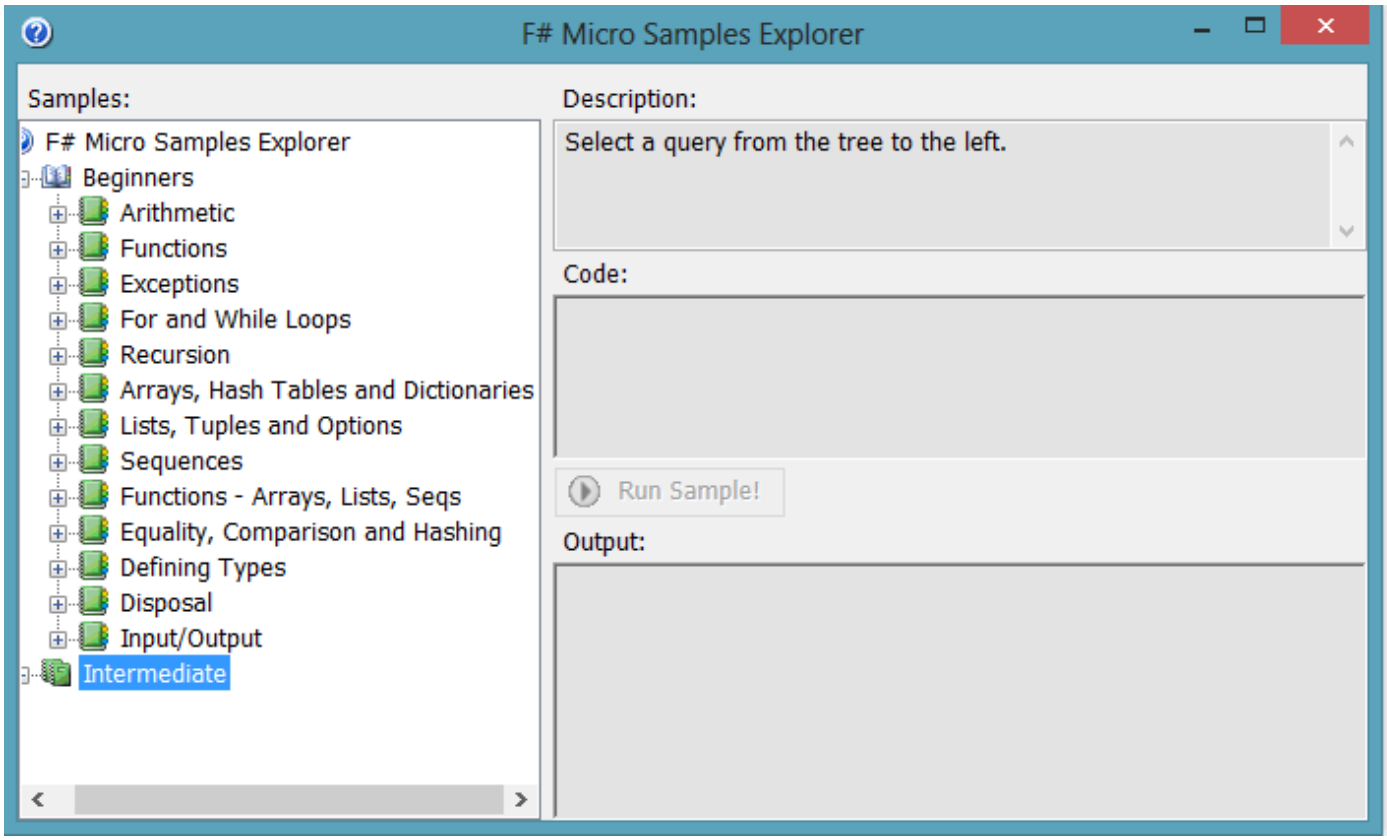
```
    ...
```

```
//Verbose
```

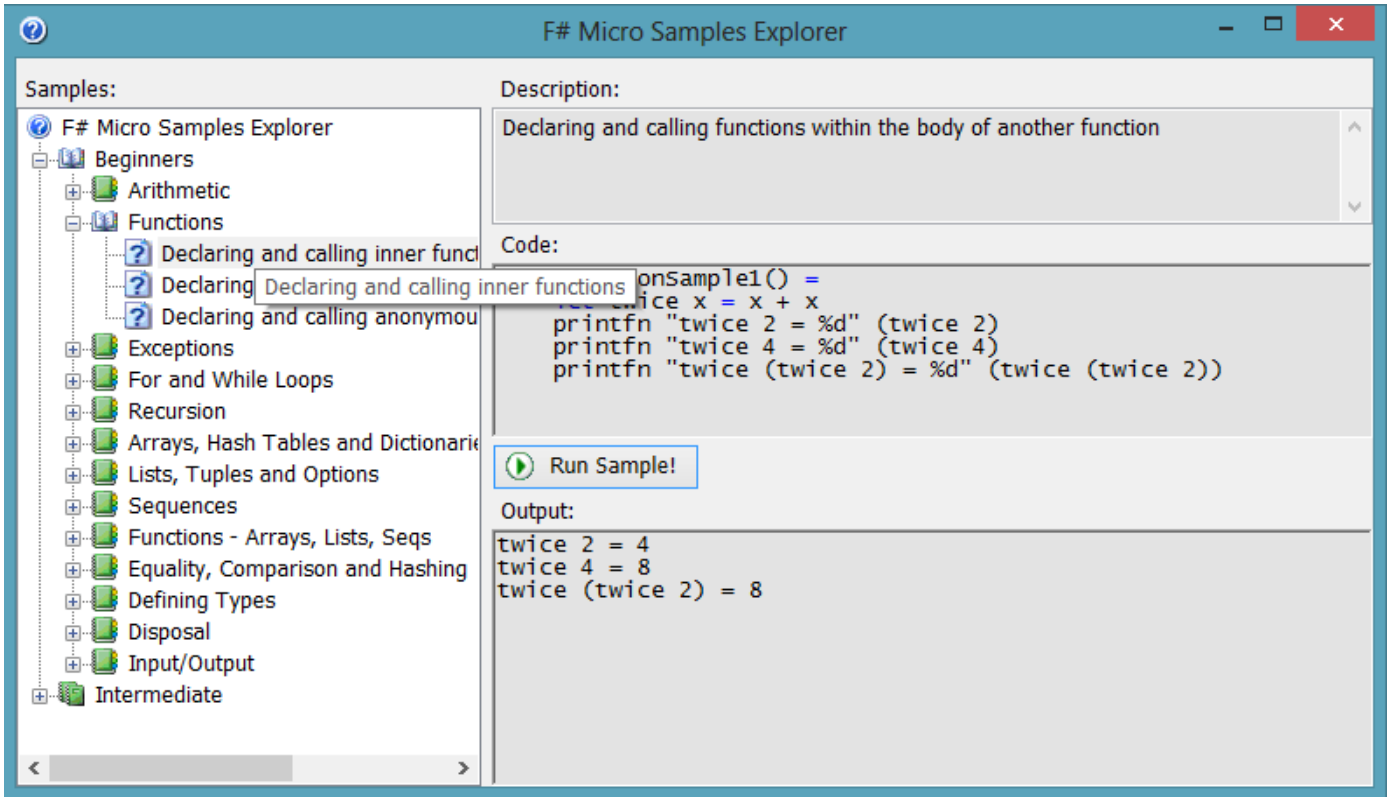
```
module <module-name> =
    begin
        ...
    end
```

عنوان: **معرفی پروژه F# Sample 101**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۲۳ ۱۲:۱۹
آدرس: www.dotnettips.info
برچسب‌ها: F#, Programming

در این پروژه که محیط آن را در تصویر زیر مشاهده می‌کنید مثال‌های متنوعی از زبان برنامه نویسی F# تعبیه شده است. از منوی سمت چپ گزینه‌ی مورد نظر خود را انتخاب کنید. کد مثال را می‌تونید در قسمت Code پروژه مشاهده کنید. با استفاده از گزینه Run Sample می‌تونید خروجی هر مثال را نیز ببینید.



برای نمونه :



دریافت سورس کامل پروژه