F#

...

F# for great good

John Patterson
in/john-patterson
johnppatterson.com

Well...

Overview of Functional Programming

- Rely on pure functions over impure functions
  - Don’t use side effects (user-input, global variables, databases, network, ...)
  - Where you have to, limit it to a small part of your architecture
- Rely on immutable data
  - Immutable objects are thread safe
  - Easier to reason about the code
    - You don’t have to hold the whole machine in your mind grapes
  - Able to be cached ‘n’ hashed without bein’ thrashed

> -1, immutable objects arent 'good'. Just more or less appropriate for specific situations. Anyone telling you one technique or another is objectively 'good' or 'bad' over another for all situations is selling you religion.
> – GrandmasterB Jun 6 '12 at 20:43

- Functions are first class citizens
  - Can be passed like any value
- Currying
  - Functions with N arguments become N composed functions
    - ```let f (x : int) (y : int) = "doggo"```
      - Type is “f : x:int -> y:int -> string”
  - Enables partial function application!
  - Happens from left to right

First parameter is actualized

```
let add a b = a + b
let addOne = add 1
addOne 2
          3
```

(add : a:int -> b:int -> int)

(addOne : int -> int)

Second becomes first parameter of addOne

- Higher-order functions
  - When you take a function as an argument or return a function from another function
  - Decorators in python!
  - `let twice f = f >> f`
    - >> is function composition, the function twice repeats a function twice
- Closure
  - The value of things in the parent scope are sealed up with a function, even after it is returned!

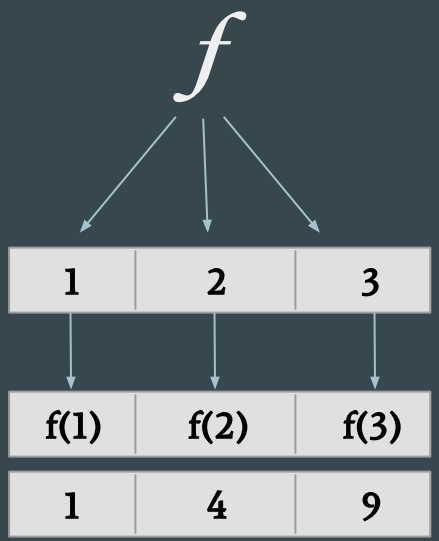
```
let getConnector apiKey =
      let apiInstance = ApiFactory(apiKey)
      let getConnectingFunction () =
            apiInstance.connect()
      getConnectingFunction
```

- Closures are a poor man's object.
- Objects are a poor man's closure.

Neither are more fundamental and each implements the other.

Functions to note!

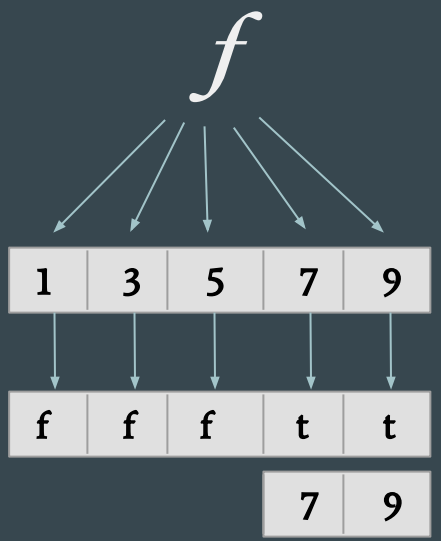
- Map
  - Takes a function and a collection and returns the collection that you get by applying the function to every element of the original collection. It's a way to transform collections
  - `Seq.map (fun x -> x * x) [1; 2; 3] = [1; 4; 9]`




```
def map_example():
    return_coll = []
    for item in [1, 2, 3]:
        return_coll.append(item * item)
    return return_coll
```

Functions to note!

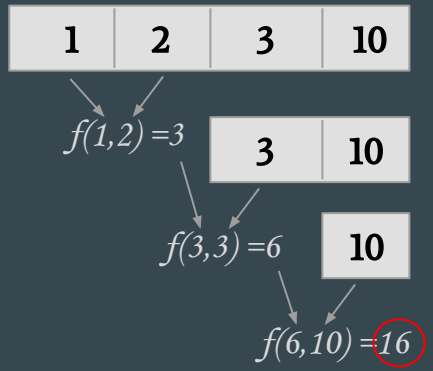
- Filter
  - Take a predicate (function returning a bool) and a collection and returns the collection of only the items your predicate is true for.
  - `Seq.filter (fun x -> x > 5) [1; 3; 5; 7; 9] = [7; 9]`




```python
def filter_example():
    return_coll = []
    for item in [1, 3, 5, 7, 9]:
        if item > 5:
            return_coll.append(item)
    return return_coll
```

Functions to note!

- Reduce
  - You get it will take a function & collection, but this *reduces* the collection to a value.
  - `Seq.reduce (fun acc x -> acc + x) [1; 2; 3; 10] = 16`
  - Your function tells it how to combine two values




```python
def reduce_example():
    partial_sum = 0
    for item in [1, 2, 3, 10]:
        partial_sum += item
    return partial_sum
```

F#

...

F# for great good

Overview

Short & Sweet: F# is like a .NET OCaml (old programming language).

- Algebraic Types
- Pattern-matching
- Easy escape hatches into mutable/imperative style
- Interops with C# and VB
- Offers functional programming without insisting upon purity
- Has some spunky little features
    - Type Providers, Active Patterns, and Computation Expressions, oh my!

Not just for Functional Programming Weenies

Let bound

You've seen these. They make variables and they bind functions. The return value of a function is the value of the last thing in the enclosing block.

```
let variableName = "hi"

let foo a b =
    printfn "%d" a
    a + b

let foo (a : int) (b : int) : int =
    printfn "%d" a
    a + b
```

```
let greet name =
    let putOnGreeting() =
        "Hello! " + name
    printfn "%s" (putOnGreeting())
```

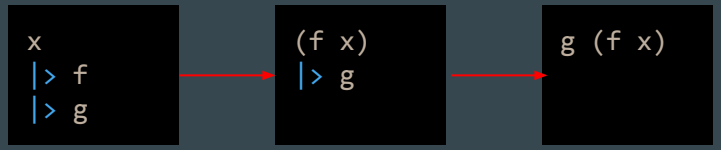
greet : string -> unit
putOnGreeting : unit -> string

Collection types

- Sequences (IEnumerable) lazily evaluated values
  - seq { ... }
  - Seq.ofList / Seq.ofArray
- Lists are linked lists
  - [“make”; “list”; “like”; “dis”; “-t”]
  - List.ofSeq / List.ofArray
- Array are normal arrays with random access
  - [| “rain”; “drop”; “drop”; “top” |]
  - Array.ofSeq / Array.ofSeq

Mappin', Filterin', 'n' Reducin'

The |> operator takes lhs and puts it on the tail end of the rhs




```
[1..100]
|> List.map ((*) 2)
|> List.filter (fun x -> x < 100 && x >= 50)
|> List.reduce (fun acc x -> acc - x)
```

```
[2..2..200]
|> List.filter (fun x -> x < 100 && x >= 50)
|> List.reduce (fun acc x -> acc - x)
```

```
[50..2..98]
|> List.reduce (fun acc x -> acc - x)
```

```
-1750
```

Tuples

- Two (or more) bits of data tied together
- Just add parens and a comma!
- Can be passed like anything else
- Type signature is T1 * T2
- Can destructure tuples returned!
- fst and snd can project the first and second field

```
let point = (1, 2)

let validate f obj : bool * int =
    let result = f obj
    if result > 5 then
        (true, result)
    else
        (false, -1)
let validateString = validate (Int32.TryParse >> snd)
let valid, value = validateString "6"
```

```
val validate f:('a -> int) -> obj:'a -> bool * int
val validateString : (string -> bool * int)
val value : int = 10
val valid : bool = true
```

Discriminated Unions & Records

Dirt cheap DSLs

```
type GarbageType =
    | RottenFood
    | Sewer
    | BourbonStreet

type BodyOdor =
    | Alright
    | Bad of GarbageType

type Person = {
    Name : string
    Age : int
    BO : BodyOdor
}
```

```
let jog person =
    {person with BO = Bad(Sewer)}

let phil = {
    Name = "Phil";
    Age = 16;
    BO = Alright
}
jog phil
```

```
val it : Person = {Name = "Phil";
                   Age = 16;
                   BO = Bad Sewer;}
```

What does this fix from C#?

```
class PaymentObject {
    public enum PaymentType { Check, Debit, Credit}
    public PaymentType Payment { get; set; }
    public string RoutingNumber { get; set; }
    public string AccountNumber { get; set; }
    public string CreditNumber { get; set; }
    public string CCV { get; set; }
    public int Pin { get; set; }
}
```

```
type Payment =
| Check of string * string
| Credit of string * string
| Debit of int
```

- C# object allows for invalid states, i.e. having a CCV and a RoutingNumber
- F# object encapsulates data within the type of the object
    - You cannot be a Debit payment with a RoutingNumber!
- Seems trivial, but for very large configurations, this is costly
- Visually less to process

Option -- A better null

- Implemented through Discriminated Unions!
- Hide data unless you have a value, no using bad values

Generic type parameter

```
type 'a Option =
| Some of 'a
| None
```

Basically exact definition of Option

```
type Person = {
    Name : string
    Age : int
    HairColor : string option
    Children : Person list option
}
```

Instead of writing Option<List<Person>>>, the order is switched and it is Person list option

Match, made in heaven

Pattern matching is a switch case on steroids!

You can match on: type, value, structure, or make your own Active Patterns!

Inexhaustive matches are a compiler error*

```
let intToString x =
    match x with
    | 0 -> "zero"
    | 1 -> "one"
    | _ -> "infinity"
```

```
let rec nextToLast arr =
    match arr with
    | [] -> None
    | [x] -> None
    | x::[y] -> Some x
    | x::xs -> nextToLast xs
```

```
open System

let parseDecimal x =
    match Decimal.TryParse(x) with
    | false, _ -> None
    | true, d -> Some d
```

* well, it's a compiler warning, but you can change a project setting to make it an error

Ultimate Cosmic Power

```
type FormStateError =
    | NoName
    | NoAge
    member x.Message =
        match x with
        | NoName -> "No name selected!"
        | NoAge -> "No age entered!"

type ValidatingResult =
    | Okay of string * int
    | Error of FormStateError
```

```
let validate model =
    if model.SelectedName = null then
        Error NoName
    elif model.EnteredAge = null then
        Error NoAge
    else
        Okay (model.SelectedName, model.EnteredAge)

let save model =
    match validate model with
    | Error kind -> invalidOp kind.Message
    | Okay (name, age) ->
        db.Insert(name, age)
```

Types vs Modules

- Module just a bunch of function together, similar to a utility class
  - module keyword either first thing in file or module <Name> = and then an indented block
  - Compile down to static classes with static methods
  - Can be nested!
  - Set of functions typically acting on a single type, may or may not contain that type

```
type PersonType = {
    First : string
    Last : string
}

module Person =
    let greet p =
        sprintf "Hi %s, %s." p.Last p.First
```

- Types are data
  - Records, DUs, etc.
  - You can extend existing types with the *with* keyword
    - Yes, even the primitives

```
type System.DateTime with
    member x.IsEvenDay =
        x.Day % 2 = 0
```

  - Functions can be attached using the static member <Foo> or member this.<Foo> syntax
  - Often dependency injection is done through type constructors

```
type IDatabase =
    abstract member FetchAll : unit -> PersonType list

type Worker(database : IDatabase) =
    member x.GreetEveryone() =
        database.FetchAll()
        |> Seq.map (fun e -> Person.greet e)
        |> Seq.iter (fun s -> printfn "%s" s)
```

Active Patterns

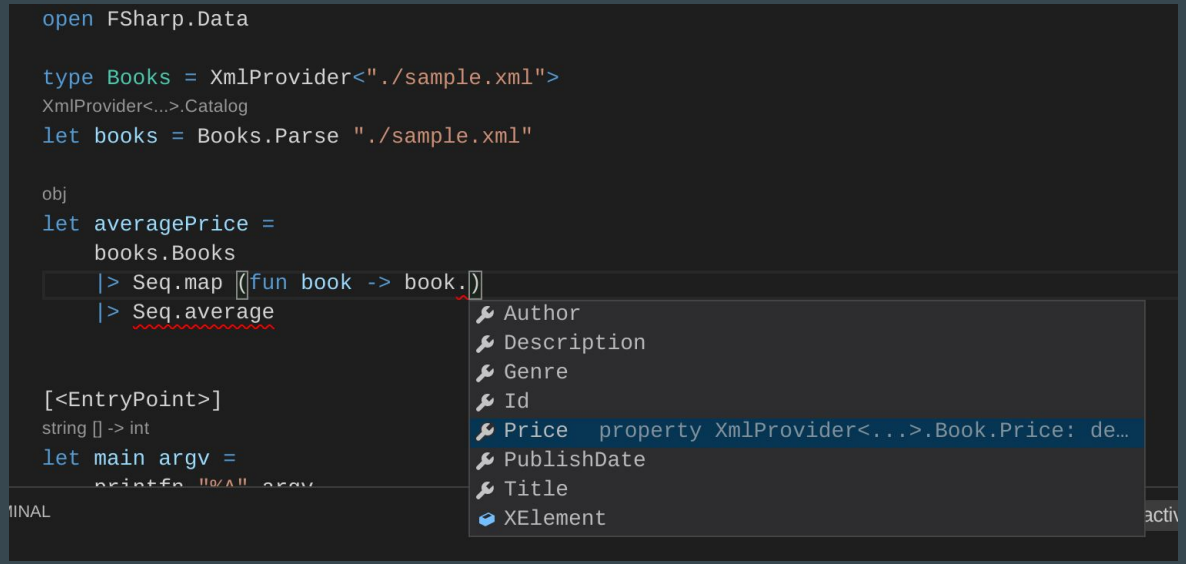
You can define your own bit of logic for match patterns. Better tryParse!

```
let tryParse f str =
    match f str with
    | (true, x) -> Some x
    | _ -> None

let (|Int|_|) = tryParse System.Int32.TryParse
let (|Bool|_|) = tryParse System.Boolean.TryParse
let (|Currency|_|) = tryParse System.Decimal.TryParse

let parse input =
    match input with
    | Int i -> printfn "Value was an int! %i" i
    | Bool b -> printfn "Value was a bool! %b" b
    | Currency c -> printfn "Value was $$$! %f" c
    | _ ->  printfn "I couldn't recognize your input: %s" input
```

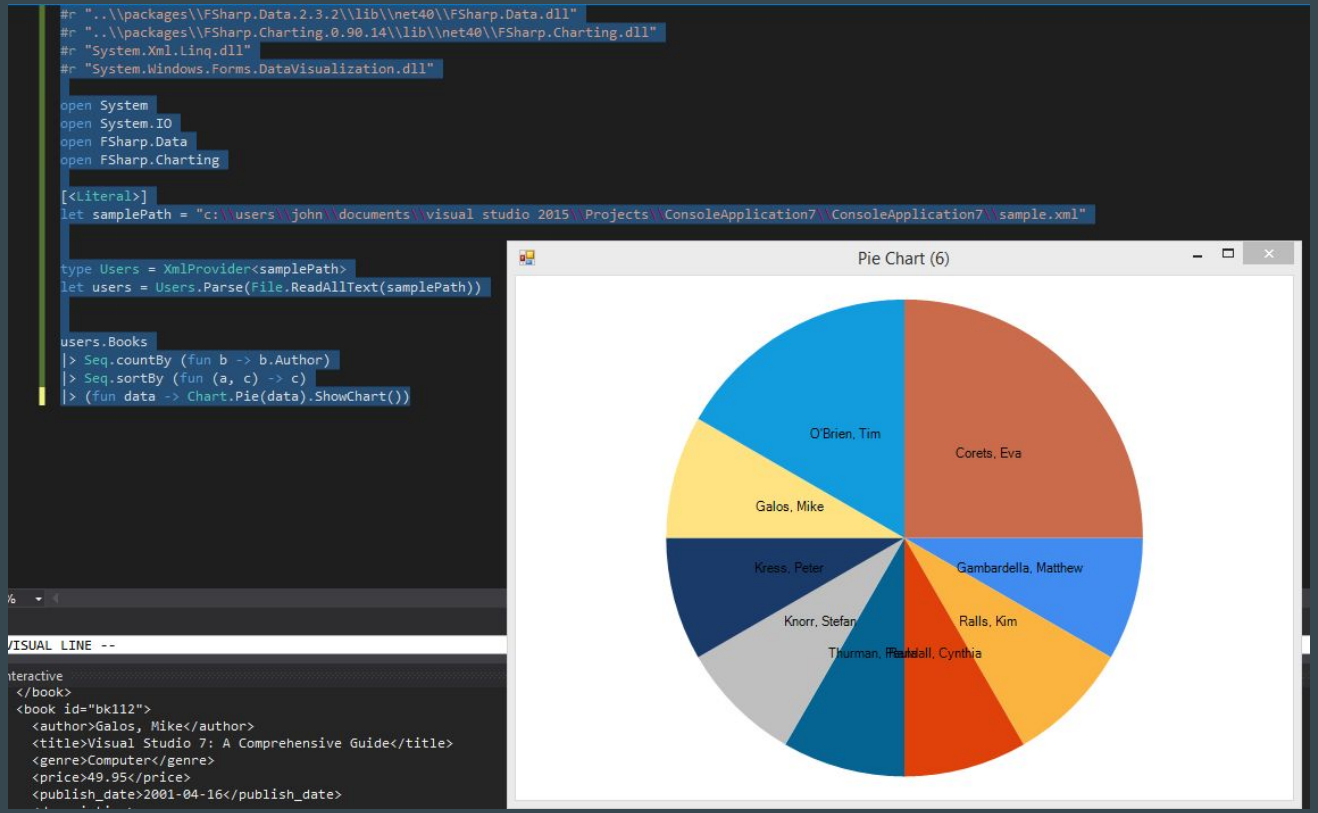
Type Providers!

Standard library that take typically untyped data: CSV, JSON, XML, etc. and transform it into a nice type that you get IntelliSense on!

```
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price>44.95</price>
    <publish_date>2000-10-01</publish_date>
    <description>An in-depth look at creating
applications with XML.</description>
  </book>
...
```



Easy (for you Windows types) Charting



Entering Weenie Exclusion Zone

Computation Expressions

Sometimes things get really gross with types.

```
let divideBy top bottom =
    match bottom with
    | 0 -> None
    | _ -> Some (top / bottom)

let inline (</>) a b = divideBy a b

let divideByWorkflow w x y z =
    match w </> x with
    | Some wx ->
        match wx </> y with
        | Some wxy ->
            wxy </> z
        | None -> None
    | None -> None
```

Computation Expressions

Just define a Monad by creating a type and supplying Bind & Return and then you get nice sugared syntax.

```
type MaybeBuilder() =
    member this.Bind(x, f) =
        match x with
        | None -> None
        | Some a -> f a
    member this.Return(x) = Some x
    member this.Delay(f) = f()

let divideBy top bottom =
    match bottom with
    | 0 -> None
    | _ -> Some (top / bottom)

let inline (</>) a b = divideBy a b
```

```
let maybe = new MaybeBuilder()
let divideByWorkflow w x y z =
    maybe {
        let! a = w </> x
        let! b = a </> y
        let! c = b </> z
        return c
    }
divideByWorkflow 6 1 3 2
divideByWorkflow 6 1 0 2

Some 1
None
```

Computation Expressions

What's happening?

```
let maybe = new MaybeBuilder()
let divideByWorkflow w x y z =
    maybe {
        let! a = w </> x
        let! b = a </> y
        let! c = b </> z
        return c
    }
```

```
let desugared w x y z =
    maybe.Delay(fun () ->
        maybe.Bind(w </> x, fun a ->
            maybe.Bind(a </> y, fun b ->
                maybe.Return(y </> z)
            )
        )
    )
```

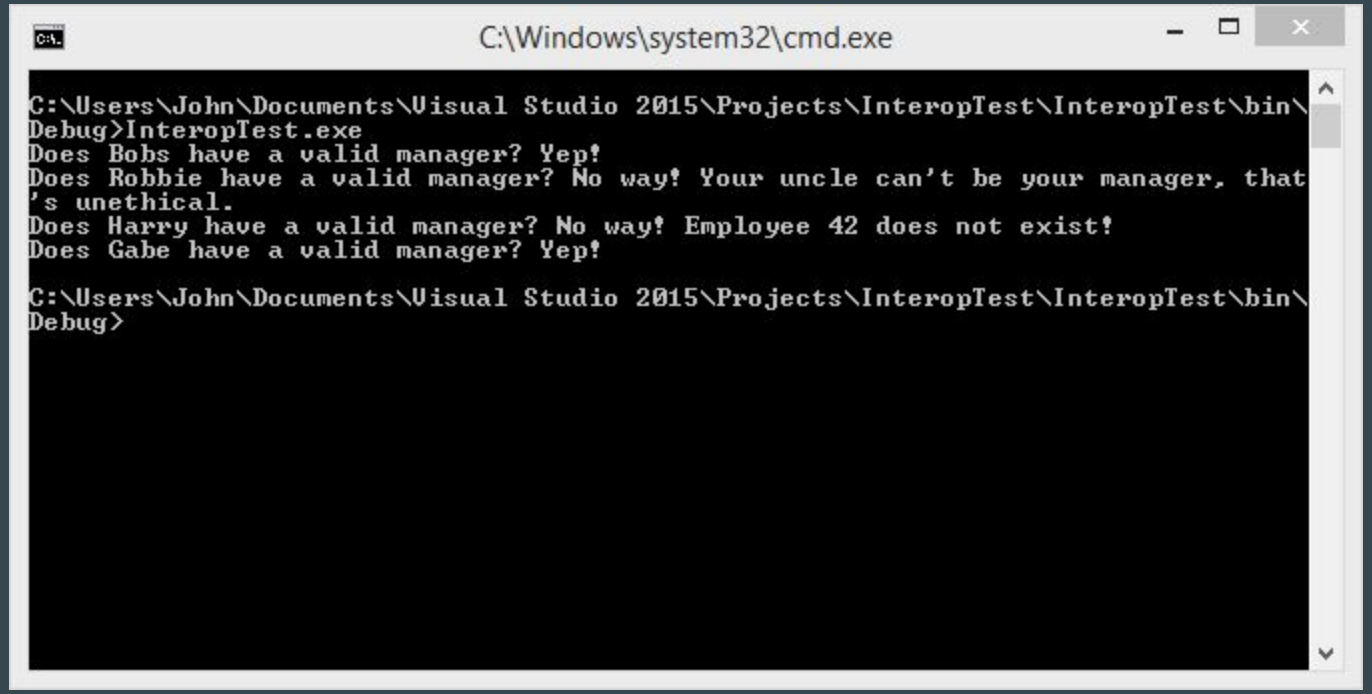
C# Interoperability

```fsharp
[<CLIMutable>]
type Employee = {
    FirstName : string
    LastName : string
    Age : int
    ID : int
    DateOfEmployment : System.DateTime
    ManagerID : int option
}

type ErrorType =
    | NoSuchEmployee of int
    | ThatsYourUncle
    with member x.Message =
            match x with
            | NoSuchEmployee id -> "Employee " + id.ToString() + " does not exist!"
            | ThatsYourUncle -> "Your uncle can't be your manager, that's unethical."

[<StructuredFormatDisplay("{ToString}")>]
type ValidationResult =
    | Okay
    | Invalid of ErrorType
    override x.ToString() =
            match x with
            | Okay -> "Yep!"
            | Invalid reason -> "No way! " + reason.Message
```

```
using System.Collections.Generic;
namespace CSharpDomain {
    public interface IDatabaseService<out T> {
        IEnumerable<T> FetchAll();
    }
}

using System;
using System.Collections.Generic;
using CSharpDomain;
using DomainTypes;
using Microsoft.FSharp.Core;
namespace DataLayer {
    public class EmployeeDataService : IDatabaseService<Employee> {
        public IEnumerable<Employee> FetchAll() {
            var uncleBob = new Employee("Bobs", "Youruncle", 73, 1, DateTime.MinValue, FSharpOption<int>.None);
            return new[] {
                uncleBob,
                new Employee("Robbie", "Rotten", 23, 2, DateTime.Now, FSharpOption<int>.Some(1)),
                new Employee("Harry", "Potter", 13, 3, DateTime.Now, FSharpOption<int>.Some(42)),
                new Employee("Gabe", "Thedog", 11, 4, DateTime.Now, FSharpOption<int>.Some(2)),
            };
        }
    }
}
```

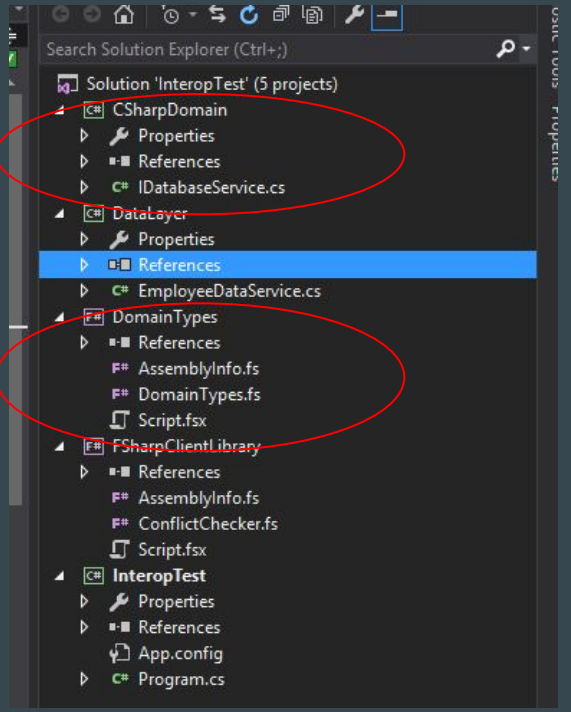
```fsharp
namespace FSharpClientLibrary
open DomainTypes
open CSharpDomain

type ConflictChecker(employeeRepo : Employee IDatabaseService) =
    let allEmployees = employeeRepo.FetchAll() |> List.ofSeq

    member x.Check e =
        match e.ManagerID with
        | None -> Okay
        | Some id ->
            match allEmployees |> List.filter (fun man -> man.ID = id) with
            | [emp] ->
                if emp.LastName.ToLower().Contains("uncle") then
                    Invalid ThatsYourUncle
                else
                    Okay
            | _ -> Invalid (NoSuchEmployee id)
```

```
using System;
using DataLayer;
using FSharpClientLibrary;

namespace InteropTest {
    internal class Program {
        private static void Main() {
            var database = new EmployeeDataService();
            var checker = new ConflictChecker(database);

            foreach (var employee in database.FetchAll()) {
                Console.WriteLine($"Does {employee.FirstName} have a valid manager? {checker.Check(employee)}");
            }
            Console.ReadKey(false);
        }
    }
}
```


C:\Windows\system32\cmd.exe
C:\Users\John\Documents\Visual Studio 2015\Projects\InteropTest\InteropTest\bin\Debug>InteropTest.exe
Does Bobs have a valid manager? Yep!
Does Robbie have a valid manager? No way! Your uncle can't be your manager, that's unethical.
Does Harry have a valid manager? No way! Employee 42 does not exist!
Does Gabe have a valid manager? Yep!
C:\Users\John\Documents\Visual Studio 2015\Projects\InteropTest\InteropTest\bin\Debug>

The Ugly Sides

```
using Microsoft.FSharp.Core;

namespace DataLayer {
    public class EmployeeDataService : IDatabaseService<Employee> {
        public IEnumerable<Employee> FetchAll() {
            var uncleBob = new Employee("Bobs", "Youruncle", 73, 1, DateTime.MinValue, FSharpOption<int>.None);
            return new[] {
```



- Taking on FSharp.Core as a dependency everywhere and you get janky FSharpOption and FSharpList, just not pretty.
- Kind of a strange project structure, often need split domain for types.
- If your C# code returns a nullable type, you lose null safety in F#

Mutable F#

In general, not encouraged, but it's simple to do.

```
let mutable x = 5.0
type Student = {
    Id : int
    Name : string
    mutable AverageGrade :  float
}

let bobby = {Id = 1; Name = "Bobby"; AverageGrade = 0.0}

let foo () =
    x <- (x + 1.0) / 2.0
    bobby.AverageGrade <- x
```

Common Tools

- Editor
  - Visual Studio (The OSS guys use Visual Studio Code)
  - Use F# Power Tools! (Also use the Ionide plugins for VS Code)
- Dependency management
  - NuGet if you're in VS
  - Paket if you're off in F# land
- Build scripting
  - FAKE or F# make if you're someone who needs to use a search engine
- Testing framework
  - fsUnit on top of xUnit
- Database access
  - SQL Type Provider (I've had some bad luck, I just interop with Dapper)
- Web framework
  - Suave

Finally,

If you need to transpile to JavaScript, there's Fable.

In conclusion

- F# is neat and cuts down on a lot of C# boilerplate
  - Things like semicolons, braces, and unnecessary type specifiers are visual noise
  - First-class record types and discriminated unions simplify common C# patterns
- F# can make your codebase safer
  - Domain driven development with Discriminated Unions can make some neat code
- It’s a low risk investment if you’re already bought into M$ stack

The F# Foundation

fsharp.org

Community for F#

c4fsharp.net

F# For Fun and Profit

fsharpforfunandprofit.com