

Introduction to F# Workshop

Exercises

Table of Contents

| | | |
|------|---------------------------|-----------|
| I. | Introduction | 2 |
| II. | Exercise 1 | 4 |
| III. | Exercise 2 | 7 |
| IV. | Exercise 3 | 10 |
| V. | Exercise 4 | 12 |
| VI. | Exercise 5 | 14 |

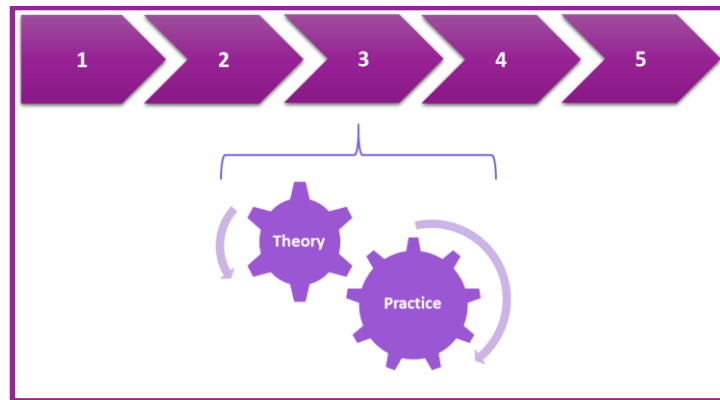
Introduction

Do you want to learn F# and Functional Programming? Well, you better start coding!

Learning a new programming language is not easy, on top of reading a lot you need to practice even more.

This workshop is designed to teach you some of the basics of F# and Functional Programming by combining theory and practice.

The course is split into 5 modules, each of them contains a presentation (theory) and one exercise (practice). You can find exercises for each module in this document, for the presentation and source code, refer to the section “Source Code, Additional Material and Updates”.




Minimum Requirements


- Visual Studio 2013
- Visual F# tools 3.1.2 or greater
- F# MVC5 Template
- XUnit Runner
- Visual F# Power Tools (optional)

Nuget Packages

- XUnit
- Unquote
- SqlProvider (TypeProvider) [alpha]
- F# Data

Code Conventions

Every time you see a box with this icon: , it means you need to run that code in the F# Interactive.

```
 > increaseCredit vipCondition customer1;;
```

When you see a white box, that is code you need to write in a source file.

```
let vipCondition customer = customer.IsVip
```

Source Code, Additional Material and Updates

<http://jorgef.github.io/fsharpworkshop/>

Author

Jorge Fioranelli (@jorgefioranelli)

Licensed under the Apache License, Version 2.0

Exercise 1

- Bindings
- Functions
- Tuples
- Records

1. Open the solution `Module1\Pre\FSharpIntro.sln` and go to the Applications project.

2. Open the file `Types.fs` and create a record type called “Customer” as follows:

```
module Types

open System

type Customer =
    { Id: int
      IsVip: bool
      Credit: float }
```

3. Send the customer type in the F# interactive by highlighting it and pressing “Alt+Enter” or right-click “Execute in Interactive” (do not highlight the module line), you should see the following output:

```
i type Customer =
    {Id: int;
     IsVip: bool;
     Credit: float;}
```


4. Create one customer in the F# interactive (press enter after each “;;”)

```
i > let customer1 = { Id = 1; IsVip = false; Credit = 10.0 };;
```


This should be the result:

```
i val customer1 : Customer = {Id = 1;
                               IsVip = false;
                               Credit = 10.0;}
```

5. Create another customer:

```
 > let customer2 = { Id = 2; IsVip = false; Credit = 0.0 };;
```

This should be the result:


```
 val customer1 : Customer = {Id = 2;  
                                IsVip = false;  
                                Credit = 0.0;}
```

6. Go to the Tests.fs file located in the Tests project, uncomment and run the test 1-1


7. Open the file Functions.cs and add a function called “tryPromoteToVip”:

```
module Functions  
  
open Types  
  
let tryPromoteToVip (customer, spendings) =  
    if spendings > 100.0 then { customer with IsVip = true }  
    else customer
```

7. Highlight the function, send it to the F# Interactive and test it by typing the following:

```
 > tryPromoteToVip (customer1, 101.0);;
```

You should see this output:

```
 val it : Customer = {Id = 1;  
                        IsVip = true;  
                        SpendingLimit = 10.0;}
```

8. Now test it with customer2

9. Uncomment and run tests 1-2 and 1-3

10. Add a function called “getSpendings” to the Functions.fs file:

```
let getSpendings customer =  
    if customer.Id % 2 = 0 then (customer, 120.0)  
    else (customer, 80.0)
```

11. Send it to the F# Interactive and test it with customer1 and customer2

12. Uncomment and run tests 1-4 and 1-5

Exercise 2

- High order functions
- Pipelining
- Partial application
- Composition

1. Without closing Visual Studio, go to File -> Close Solution and then to File -> Open Project/Solution... and select Module2\Pre\FSharpIntro.sln and go to the Applications project. You need to keep Visual Studio open so you don't lose the F# Interactive session.

2. Create a function called "increaseCredit" in the file Functions.fs:


```
let increaseLimit customer =  
    if customer.IsVip then { customer with Credit = customer.Credit + 100.0 }  
    else { customer with Credit = customer.Credit + 50.0 }
```

3. Send it to the F# Interactive and test it with customer1 and customer2.

4. Refactor "increaseCredit" to be able receive the condition as a parameter:

```
let increaseCredit condition customer =  
    if condition customer then { customer with Credit = customer.Credit + 100.0 }  
    else { customer with Credit = customer.Credit + 50.0 }
```

5. Send the function to the F# Interactive and test it using a lambda expression in this way:


```
 > increaseLimit (fun c -> c.IsVip) customer1;;
```

6. Uncomment and run tests 2-1, 2-2 and 2-3


7. Create a function called "vipCondition" in the file Functions.fs:

```
let vipCondition customer = customer.IsVip
```



8. Send the function to the F# Interactive and test the “increaseCredit” function again but this time using the “vipCondition” function:

```
 > increaseCredit vipCondition customer1;;
```

9. Now test it again but this time using the pipelining operator to:

```
 customer1 |> increaseCredit vipCondition;;
```

10. Try calling “increaseCredit” with just “vipCondition” and check the result is another function that expected the missing argument (customer):

```
 > increaseLimit increaseCondition;;  
val it : (Customer -> Customer) = <fun:it@3-5>
```

11. Uncomment and run tests 2-4 and 2-5

12. Create a function called “increaseCreditUsingVip” in the file Functions.fs:

```
let increaseCreditUsingVip = increaseCredit vipCondition
```

13. Send it to the F# Interactive and test it

14. Uncomment and run test 2-6

15. Create a function called “upgradeCustomer” in the file Functions.fs:

```
let upgradeCustomer customer =  
    let customerWithSpending = getSpending customer  
    let promotedCustomer = tryPromoteToVip customerWithSpending  
    let upgradedCustomer = increaseCreditUsingVip promotedCustomer  
    upgradedCustomer
```

16. Send “upgradeCustomer” to the F# Interactive and test it with customer1 and customer2

17. Refactor “upgradeCustomer” to use the pipelining operator and test it in the F# interactive:

```
let upgradeCustomer customer =  
    customer  
    |> getSpending  
    |> tryPromoteToVip  
    |> increaseCreditUsingVip
```

18. Send “upgradeCustomer” to the F# Interactive and test it with customer1 and customer2

19. Refactor “upgradeCustomer” again to use composition:

```
let upgradeCustomer = getSpending >> tryPromoteToVip >> increaseCreditUsingVip
```

20. Send “upgradeCustomer” to the F# Interactive and test it with customer1 and customer2

21. Uncomment and run tests 2-7 and 2-8

Exercise 3

- Options
- Pattern matching
- Discriminated unions
- Units of measure

1. Without closing Visual Studio, go to File -> Close Solution and then to File -> Open Project/Solution... and select Module3\Pre\FSharpIntro.sln and go to the Applications project. You need to keep Visual Studio open so you don't lose the F# Interactive session.

2. Create a new record called "PersonalDetails", a discriminated union called "Notifications" and two units of measure "AUD" and "USD". Add them to the "Customer" in the file Types.fs (note that you need to declare them before "Customer"):

```
type PersonalDetails =  
    { FirstName: string  
      LastName: string  
      DateOfBirth: DateTime }  
  
[<Measure>] type AUD  
[<Measure>] type USD  
  
type Notifications =  
    | NoNotifications  
    | ReceiveNotifications of receiveDeals: bool * receiveAlerts: bool  
  
type Customer =  
    { Id: int  
      IsVip: bool  
      Credit: float<USD>  
      PersonalDetails: PersonalDetails option  
      Notifications: Notifications }
```

3. Send the tree types to the F# Interactive and create new instances of "Customer" as the type changed (if you receive the error: The type 'DateTime' is not defined, type "open System;;" in the F# Interactive and try again).

4. Open the file Data.fs, uncomment and send each customer to the F# Interactive.

5. Update the "increaseCredit" function to use USD:

```
let increaseCredit condition customer =  
    if condition customer then { customer with Credit = customer.Credit + 100.0<USD> }  
    else { customer with Credit = customer.Credit + 50.0<USD> }
```

6. Uncomment and run tests 3-1 and 3-2 (you will also need to uncomment the “customer” value defined at the top of the file Test.fs)

7. Create a function called “isAdult” in the file Functions.fs:

```
let isAdult customer =  
    match customer.PersonalDetails with  
    | None -> false  
    | Some d -> d.DateOfBirth.AddYears(18) <= DateTime.Now.Date
```

8. Send “isAdult” to the F# Interactive and test it with customer1 and customer2.

9. Uncomment and run tests 3-3, 3-4 and 3-5

10. Create a function called “getAlert” in the file Functions.fs:

```
let getAlert customer =  
    match customer.Notifications with  
    | ReceiveNotifications(receiveDeals = _; receiveAlerts = true) ->  
        Some (sprintf "Alert for customer: %i" customer.Id)  
    | _ -> None
```

11. Send “getAlert” to the F# Interactive and test it with customer1 and customer2.

12. Uncomment and run tests 3-6 and 3-7

Exercise 4

- Functional lists
- Recursion
- List module

1. Without closing Visual Studio, go to File -> Close Solution and then to File -> Open Project/Solution... and select Module4\Pre\FSharpIntro.sln and go to the Applications project. You need to keep Visual Studio open so you don't lose the F# Interactive session.

2. Create a new function called "getSpendingByMonth" in the file Types.fs right before the "getSpending" function:

```
let tryPromoteToVip (customer, spendings) = ...

let getSpendingByMonth customer =
    [for _ in [1..12] do
        if customer.Id % 2 = 0 then yield 150.0
        else yield 60.0]

let getSpending customer = ...
```

4. Send it to the F# Interactive and test it with customer1 and customer2

5. Uncomment and run tests 4-1 and 4-2

6. Create another function called "weightedMean" right after the "getSpendingByMonth":

```
let getSpendingByMonth customer = ...

let weightedMean values =
    let rec recursiveWeightedMean items accumulator =
        match items with
        | [] -> accumulator / float (List.length values)
        |(w,v)::vs -> recursiveWeightedMean vs (w * v)
    recursiveWeightedMean values 0.0

let getSpending customer = ...
```

7. Uncomment and run test 4-3

8. Change the implementation of "getSpending" to use "getSpendingByMonth" and "weightedMean":

```
let getSpending customer =  
    let weights = [0.8; 0.9; 1.0; 0.7; 0.9; 1.0; 0.8; 1.0; 1.0; 1.0; 0.8; 0.7]  
    let spending = customer  
        |> getSpendingByMonth  
        |> List.zip weights  
        |> weightedMean  
    (customer, spending)
```

9. Send both “weightedMean” and “getSpending” to the F# Interactive and test “getSpending” with customer1 and customer2

10. Uncomment and run test 4-4

Exercise 5

- Object Oriented Programming
- Type Providers
- Web Applications

1. Without closing Visual Studio, go to File -> Close Solution and then to File -> Open Project/Solution... and select Module5\Pre\FSharpIntro.sln and go to the Applications project. You need to keep Visual Studio open so you don't lose the F# Interactive session.

2. Add the following Nuget packages (use the option include pre-releases when searching)

- SqlProvider (Type providers for SQL Server access) v0.0.0-alpha (Prerelease)

- F# Data (Library of F# type providers and data access tools) v2.0.14

After installing each package you will see a security dialog asking you if you want to enable the type provider, click "Enable".

3. Create a database called "FSharpIntro" in your local SqlServer and run the Data.sql file located in the Application project.

4. Open the Data.fs file and add the following code (adjust the connection string to point to your local SQLServer):

```
open FSharp.Data
open FSharp.Data.Sql
open FSharp.Data.Sql.Common

type db = SqlDataProvider<ConnectionString = "Data Source=.;
                                           Initial Catalog=FSharpIntro;
                                           Integrated Security=SSPI;",
                                           DatabaseVendor = DatabaseProviderTypes.MSSQLSERVER,
                                           IndividualsAmount = 1000,
                                           UseOptionTypes = true>
```


5. Right after that, write the following "getCustomers" function:

```
let getCustomers () =
    let ctx = db.GetDataContext()
    ctx.[dbo].[Customers]
    |> Seq.map (fun c ->
        { Id = c.Id
          IsVip = c.IsVip
          Credit = c.Credit * 1.0<USD>
          PersonalDetails = None
          Notifications = NoNotifications })
```


6. Right click on the “References” folder of the Application project and select “Send References to F# Interactive”.

7. Highlight everything in the Data.fs file but the first two lines (“module Data” and “open Types” should not be included) and executed in the F# Interactive.

8. Execute the following code in the F# Interactive:

```
 > getCustomers ();;
```

9. You should get the following result:

```
 val it : seq<Customer> =  
    seq  
        [{Id = 1;  
          IsVip = false;  
          Credit = 0.0;  
          PersonalDetails = null;  
          Notifications = NoNotifications;}; {Id = 2;  
                                              IsVip = false;  
                                              Credit = 10.0;  
                                              PersonalDetails = null;  
                                              Notifications = NoNotifications;};  
        {Id = 3;  
          IsVip = false;  
          Credit = 30.0;  
          PersonalDetails = null;  
          Notifications = NoNotifications;}; {Id = 4;  
                                              IsVip = true;  
                                              Credit = 50.0;  
                                              PersonalDetails = null;  
                                              Notifications = NoNotifications;}]
```


6. The write the following “updateCustomer” function right after the “getCustomers” one:

```
let updateCustomer customer =  
    let ctx = db.GetDataContext()  
    let dbCustomer = query { for c in ctx.[dbo].[Customers] do  
                            where (c.Id = customer.Id)  
                            select c }  
                            |> Seq.exactlyOne  
    dbCustomer.IsVip <- customer.IsVip  
    dbCustomer.Credit <- customer.Credit / 1.0<USD>  
    ctx.SubmitUpdates ()
```

7. Open the file Services.fs and add the following class:

```
type CustomerService() =  
  
    member this.GetCustomers () = Data.getCustomers ()  
  
    member this.UpgradeCustomers (ids: int seq) =  
        let customers = Data.getCustomers ()  
        ids  
        |> Seq.map (fun id -> customers  
                    |> Seq.find (fun c -> c.Id = id))  
        |> Seq.map (Functions.upgradeCustomer >> Data.updateCustomer)  
        |> List.ofSeq  
        |> ignore
```

8. Open the file CustomersController.cs located in the CSharpWeb project, and add the following code:

```
using System.Linq;
using System.Web.Mvc;
using CSharpWeb.Models;
using Services;

namespace CSharpWeb.Controllers
{
    public class CustomersController : Controller
    {
        private readonly CustomerService service;

        public CustomersController()
        {
            service = new CustomerService();
        }

        public ActionResult Index()
        {
            var customers = service.GetCustomers();
            var model = customers.Select(c => new CustomerViewModel
            {
                Id = c.Id,
                Credit = c.Credit,
                IsVip = c.IsVip
            });
            return View(model);
        }

        [HttpPost]
        public ActionResult Post(int[] ids)
        {
            service.UpgradeCustomers(ids);
            return RedirectToAction("Index");
        }
    }
}
```

9. Set CSharpWeb as the start project and run it. Navigate to /Customers, select some customers and press "Upgrade"

10. Open the Data.fs file located in the Application project and add the following code at the end:

```
type json = JsonProvider<"Data.json">

let getSpending id =
    json.Load("App_Data\Data.json")
    |> Seq.filter (fun c -> c.Id = id)
    |> Seq.collect (fun c -> c.Spending
        |> Seq.map (fun s -> float(s)))
    |> List.ofSeq
```

11. Change the “getSpendingsByMonth” function located in the file Functions.fs:

```
let getSpendingsByMonth customer = customer.Id |> Data.getSpendings
```

12. Run the CSharpWeb project again, navigate to /Customers, select some customers and press “Upgrade”

13. Go to File -> Close Solution and then to File -> Open Project/Solution... and select Module5\Post\FSharpIntro.sln and explore the new FSharpWeb project, it is a full F# MVC5 project.

14. Set FSharpWeb as start project and run it. Go to /Customers, select some customers and press “Upgrade”.