F# Introduction Workshop

Exercises

Table of Contents

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 4 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".
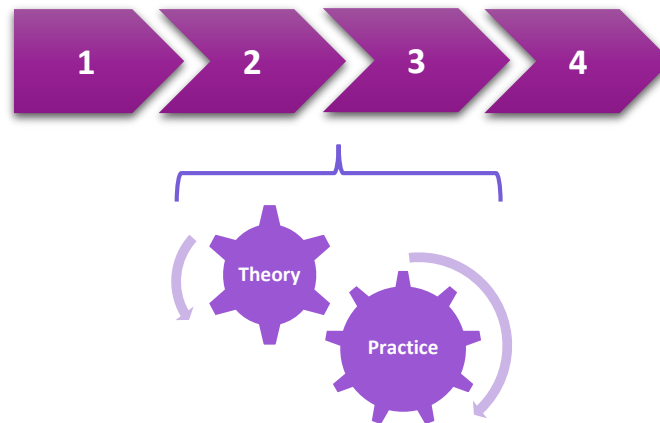
Minimum Requirements

- Visual Studio 2013 Professional or higher
- Visual F# tools 3.1.2 or higher
- Visual F# Power Tools (optional)

Nuget Packages

- XUnit
- XUnit Visual Studio Runner
- Unquote
- F# Data (Id: FSharp.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, this is code you need to write in a source file.

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

Source Code, Additional Material and Updates

http://fsharpworkshop.com/

https://github.com/jorgef/fsharpworkshop

Author

Jorge Fioranelli (@jorgefioranelli)

Licensed under the Apache License, Version 2.0

Exercise 1

- Bindings
- Functions
- Tuples
- Records

Duration: 15 minutes

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

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 Types" line), you should see the following output:

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

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

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

This should be the result:

```
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 customer2 : Customer = {Id = 2;
                            IsVip = false;
                            Credit = 0.0;}
```

6. Go to the Tests.fs file located in the Tests project, uncomment, compile and run the test 1-1 (use the Test Explorer to run it, it may take a few seconds to appear in the list).

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
```

8. Highlight the function (without including "module Functions" and "open Types" lines), 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;
                     Credit = 10.0;}
```

Now test it with customer2.

9. Uncomment, compile 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, compile and run tests 1-4 and 1-5.

Exercise 2

- High order functions
- Pipelining
- Partial application
- Composition

Duration: 20 minutes

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 increaseCredit 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. Change "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:

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

6. Uncomment, compile 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 if the result is another function that expects the missing argument (customer):

```
> increaseCredit vipCondition;;
val it : (Customer -> Customer) = <fun:it@5-4>
```

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

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

```
let increaseCreditUsingVip = increaseCredit vipCondition
```

13. Uncomment, compile and run test 2-6

14. Create a function called "upgradeCustomer" in the file Functions.fs:

```
let upgradeCustomer customer =
    let customerWithSpendings = getSpendings customer
    let promotedCustomer = tryPromoteToVip customerWithSpendings
    let upgradedCustomer = increaseCreditUsingVip promotedCustomer
    upgradedCustomer
```

15. Send "increaseCreditUsingVip" and "upgradeCustomer" to the F# Interactive and test "upgradeCustomer" with customer1 and customer2.

16. Refactor "upgradeCustomer" to use the pipelining operator and test it in the F# interactive:

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

17. Send the new "upgradeCustomer" to the F# Interactive and test it with customer1 and customer2.

18. Refactor "upgradeCustomer" again to use composition:

```
let upgradeCustomer = getSpendings >> tryPromoteToVip >> increaseCreditUsingVip
```

19. Uncomment, compile and run tests 2-7 and 2-8.

Exercise 3

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

Duration: 20 minutes

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". You need to add them to the "Customer" in the file Types.fs (note that they need to be declared before "Customer"):

```fsharp
module Types

open System

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. Highlight all but the "module Types" line and send it to the F# Interactive (include "open System").

4. Open the file Data.fs, uncomment both customers and send them to the F# Interactive (do not select the "module …" and "open …: lines).

5. Update the "increaseCredit" function to use USD in the file Functions.fs:

```
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, compile 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, compile 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, compile and run tests 3-6 and 3-7.

Exercise 4

- Functional lists
- Recursion
- Object-oriented programming
- Type providers

Duration: 20 minutes

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.

When you open or compile the solution you will see a security dialog asking you to enable the type provider, click "Enable".

2. Open the Data.fs file located in the Application project and add the following code:

```fsharp
open Types
open FSharp.Data

type Json = JsonProvider<"Data.json">

let getSpendings id =
    Json.Load "Data.json"
    |> Seq.filter (fun c -> c.Id = id)
    |> Seq.collect (fun c -> c.Spendings |> Seq.map float)
    |> List.ofSeq
```

3. Create a new function called "getSpendingsByMonth" in the file Functions.fs right after "tryPromoteToVip" and before "getSpendings":

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

4. Uncomment, compile and run test 4-1.

5. Create another function called "weightedMean" right after the "getSpendingsByMonth":

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

6. Uncomment, compile and run test 4-2.

7. Change the implementation of "getSpendings" to use "getSpendingsByMonth" and "weightedMean":

```
let getSpendings 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
                    |> getSpendingsByMonth
                    |> List.zip weights
                    |> weightedMean
    (customer, spending)
```

8. Uncomment, compile and run test 4-3.

9. Open the Data.fs file and add the following code:

```
type Csv = CsvProvider<"Data.csv">

let getCustomers () =
    let file = Csv.Load "Data.csv"
    file.Rows
    |> Seq.map (fun c ->
        { Id = c.Id
          IsVip = c.IsVip
          Credit = float c.Credit * 1.0<USD>
          PersonalDetails = None
          Notifications = NoNotifications })
```

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

```
type CustomerService() =
    member this.GetCustomers () = Data.getCustomers ()
    member this.UpgradeCustomer id =
        Data.getCustomers ()
        |> Seq.find (fun c -> c.Id = id)
        |> Functions.upgradeCustomer
```

11. Uncomment, compile and run tests 4-4 and 4-5.

12. Open Program.fs, uncomment all the code and run the application