F# Workshop

Exercises

Table of Contents

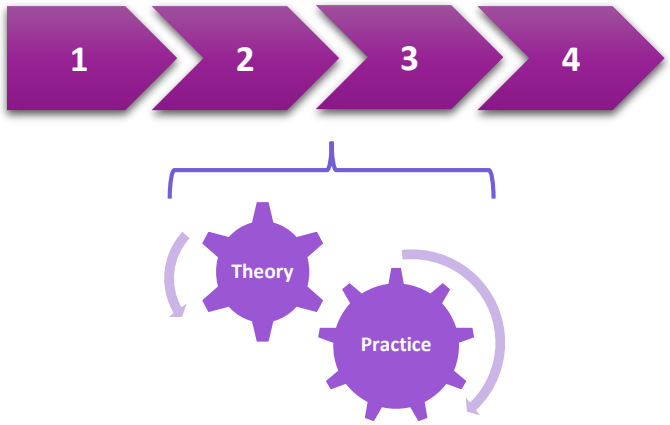
Introduction

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

You can use one of the following editors:

- Visual Studio 2015 Community Edition or higher (Win), with:
  - Visual F# tools 4.0 or higher
  - Paket for Visual Studio
  - Visual F# Power Tools (optional)
- Visual Studio Code 0.10.6 or higher (Win, Mac or Linux), with:
  - Visual F# tools 4.0 or higher
  - Ionide-fsharp extension
    - Add C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0 to the PATH environment variable.

You also need internet connection to download the dependencies.

Dependencies

- XUnit
- Unquote
- 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, 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)

Module 1

- Bindings
- Functions
- Tuples
- Records

Duration: 15 minutes

> Do not copy and paste the code, you must type each exercise in, manually.

Before we start

Visual Studio Users

- Open Visual Studio
- Open the solution FSharpWorkshop.sln located in the root folder.
- Build the solution (Build -> Build Solution). This process will download all the packages and will prompt a security dialog asking you to enable the type provider, click “Enable”.
- Double check that the build finishes successfully.
- Open the F# Interactive (View -> Other Windows -> F# Interactive)

Visual Studio Code Users

- Open Visual Studio Code
- Open the root folder (File -> Open Folder)
- Open the F# Interactive (View -> Command Palette -> FSI: Start)
- Open the console and go to the Module1 folder and execute runtests.bat (Win) / runtests.sh (Mac or Linux). This process will compile and download all the packages (no tests are enabled yet).
- Double check it finishes without errors.

1. Go to the Module1/Application and open the file Types.fs and create a record type called “Customer” as follows:

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

2. Save the Types.fs file and 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: decimal;}
```

3. Open Try.fsx file and create a new customer as follows and send it to the F# Interactive.

```
let customer1 = { Id = 1; IsVip = false; Credit = 10M }
```

This should be the result:

```
val customer1 : Customer = {Id = 1;
                            IsVip = false;
                            Credit = 10M;}
```

4. Create another customer and send it to the F# Interactive.

```
let customer2 = { Id = 2; IsVip = false; Credit = 0M }
```

This should be the result:

```
val customer2 : Customer = {Id = 2;
                            IsVip = false;
                            Credit = 0M;}
```

5. Go to the Module1/Tests/Tests.fs and uncomment the test 1-1 and save the file. Execute runtests.bat (Win) / runtests.sh (mac or Linux) from the console and check the test passes.

6. Open the file Functions.fs and add a function called “tryPromoteToVip”:

```
let tryPromoteToVip (customer, spendings) =
    if spendings > 100M then { customer with IsVip = true }
    else customer
```

7. Highlight the function (without including “module Functions” and “open Types” lines) and send it to the F# Interactive. You should see this output:

```
val tryPromoteToVip : customer:Customer * spendings:decimal -> Customer
```

8. Go to the Try.fsx file, invoke the new function and send it to the F# Interactive

```
let vipCustomer1 = tryPromoteToVip (customer1, 101M)
```

You should see this output:

```
val vipCustomer : Customer = {Id = 1;
                              IsVip = true;
                              Credit = 10M;}
```

Now test it with customer2 using 99M as spendings in the Try.fsx file.

9. Uncomment tests 1-2 and 1-3, save all the files and run the tests.

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

```
let getSpendings customer =
    if customer.Id % 2 = 0 then (customer, 120M)
    else (customer, 80M)
```

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

12. Uncomment tests 1-4 and 1-5, save all the files and run the tests.

Module 2

- High order functions
- Pipelining
- Partial application
- Composition

Do not copy and paste the code, you must type each exercise in, manually.

Duration: 20 minutes

1. Go to the Module2/Application, open Functions.fs and create a function called “increaseCredit”:

```
let increaseCredit customer =
    if customer.IsVip then { customer with Credit = customer.Credit + 100M }
    else { customer with Credit = customer.Credit + 50M }
```

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 + 100M }
    else { customer with Credit = customer.Credit + 50M }
```

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.

Module 3

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

Do not copy and paste the code, you must type each exercise in, manually.

Duration: 20 minutes

1. Go to the Module3/Application project.

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”):

```
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: decimal<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 + 100M<USD> }
  else { customer with Credit = customer.Credit + 50M<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.

Module 4

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

Do not copy and paste the code, you must type each exercise in, manually.

Duration: 20 minutes

1. Go to the Module4/Application project.

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

```
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)
    |> List.ofSeq
```

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

```
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 / (decimal (List.length values))
        | (w,v)::vs -> recursiveWeightedMean vs (accumulator + w * v)
    recursiveWeightedMean values 0M
```

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

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

```
let getSpendings customer =
    let weights = [0.8M; 0.9M; 1M; 0.7M; 0.9M; 1M; 0.8M; 1M; 1M; 1M; 0.8M; 0.7M]
    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 * 1M<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