F# Workshop

Exercises

v2.2 May-2016

Table of Contents

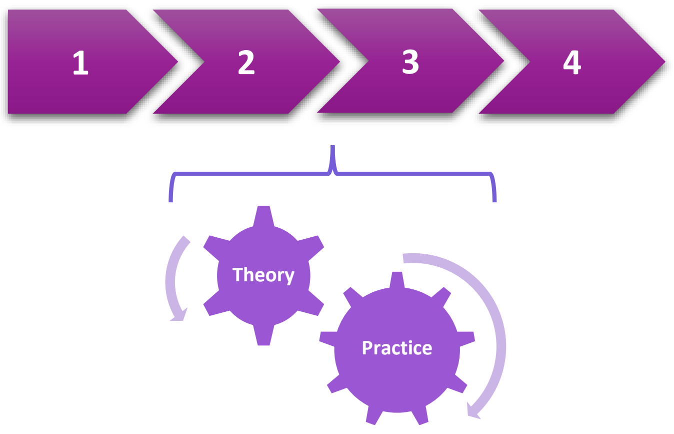
Introduction

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".



Pre-requisites

Windows

- Visual Studio 2015 Community or
- Xamarin Studio 6 or
- Atom + F# Compiler + Ionide package or
- Visual Studio Code + F# Compiler + Ionide package

Mac

- Xamarin Studio 6 + Mono or
- Atom + Mono + Ionide package or
- Visual Studio Code + Mono + Ionide package

Linux

- Atom + Mono + Ionide package or
- Visual Studio Code + Mono + Ionide package

You also need internet connection to download the dependencies.

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)

Before we start

Visual Studio Users (Windows)

- 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 if it is not open (View -> Other Windows -> F# Interactive)
- Go to the Module1/Application, open Try.fsx, write “let a = 1”, highlight the entire line, right click and select “Execute in Interactive”.
- Double check you see “val a : int = 1” in the F# Interactive window.

Xamarin Studio Users (Mac)

- Open the Terminal, go to the Module1 folder and run ./runtests.sh. If you get “Permission Denied” run chmod +x runtests.sh and try again (you will need to do the same for all the other .sh files).
- Double check that the build finishes successfully.
- Open Xamarin Studio
- Open the solution FSharpWorkshop.sln located in the root folder.
- Open the F# Interactive if it is not open (View -> Pads -> F# Interactive)
- Go to the Module1/Application, open Try.fsx, write “let a = 1”, highlight the entire line, right click and select “Send selection to F# Interactive”.
- Double check you see “val a : int = 1” in the F# Interactive window.

Atom / Visual Studio Code Users (Windows, Mac or Linux)

- Open the Command Prompt (Windows) / Terminal (Mac or Linux), go to the Module1 folder and execute runtests.bat (Windows) / runtests.sh (Mac or Linux). This process will compile and download all the packages (no tests are enabled yet). If you get “Permission Denied” run chmod +x runtests.sh and try again (you will need to do the same for all the other .sh files).
- Double check it finishes without errors.
- Open Visual Studio Code
- Open the root folder (File -> Open Folder)
- Open the F# Interactive (View -> Command Palette -> FSI: Start)
- Go to the Module1/Application, open Try.fsx, write “let a = 1”, highlight the entire line and go to View -> Command Palette -> FSI: Send Selection.
- Double check you see “val a : int = 1” in the F# Interactive window.

Module 1

- Bindings
- Functions
- Tuples
- Records

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

Duration: 20 minutes

Step 1: Create a Customer type

1.1. Go to the Module1/Application, open Types.fs and create a record type called "Customer" with the following fields:

- Id: int
- IsVip: bool
- Credit: decimal

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

1.2. Execute the customer type in the F# interactive (do not highlight the "module Types" line). For more details about how to execute code in the F# Interactive see the "Before we start" section. You should see the following output:

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

1.3. Open Module1/Application/Try.fsx, create a new Customer called customer1 and execute it in the F# Interactive. Use the following values:

- Id = 1
- IsVip = false
- Credit = 10M

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

This should be the result:

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

1.4. Create another Customer called customer2 and execute it in the F# Interactive. Use the following values:

- Id = 2
- IsVip = false
- Credit = 0M

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

This should be the result:

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

1.5. Open Module1/Tests/Tests.fs, uncomment the test 1-1, save all the files, go to the Command Prompt/Terminal and run the tests by executing runtests.bat (Win) or runtests.sh (Mac or Linux) located in the Module1 folder.

Step 2: Create a tryPromoteToVip function

2.1. Open the file Module1/Application/Functions.fs and add a function called "tryPromoteToVip" that

- Receives as parameter the customer and his/her spendings as a tuple: (customer, spendings)
- Returns the customer with Vip = true only if the spendings are greather than 100M

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

2.2. Highlight the function (without including "module Functions" and "open Types" lines) and execute it in the F# Interactive. You should see this output:

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

2.3. Open Module1/Application/Try.fsx, invoke the tryPromoteToVip function and assign the result to a value called vipCustomer1. Then execute it in the F# Interactive. Use the following values:

- (customer1, 101M)

```
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 Module1/Application/Try.fsx file.

2.4. Open Module1/Tests/Tests.fs, uncomment tests 1-2 and 1-3, save all the files and run the tests.

Step 3: Create a getSpendings function

3.1. Add a function called "getSpendings" to Module1/Application/Functions.fs that

- Receives a customer as parameter
- Returns a tuple with the customer and its spendings, following these rules:
    - If customer.Id is divisible by 2, return spendings = 120M
    - If customer.Id is not divisible by 2, return spendings = 80M

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

3.2. Execute getSpendings in the F# Interactive and test it with customer1 and customer2 in Module1/Application/Try.fsx.

3.3. Open Module1/Tests/Tests.fs, 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

Step 1: Create an increaseCredit function

1.1. Add a function called “increaseCredit” to Module2/Application/Functions.fs that

- Receives a customer as parameter
- Returns a customer with extra credit, following these rules
    - If customer.IsVip is true, return an additional 100M of credit
    - If customer.IsVip is false, return an additional 50M of credit

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

1.2. Execute it in the F# Interactive and test it with customer1 and customer2 in Module2/Application/Try.fsx.

Step 2: Extract the condition from the increaseCredit function

2.1. Change “increaseCredit” such as:

- Receives the condition to evaluate as first parameter
- Receives the customer as second parameter
- Returns a customer with extra credit, following these rules
    - If the result of evaluating the condition with the customer is true, return an additional 100M of credit
    - If the result of the condition evaluation is false, return an additional 50M of credit

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

2.2. Execute the function in the F# Interactive and test it in Module2/Application/Try.fsx using customer1 and a lambda that evaluates whether the customer is VIP or not.

```
let customer1WithMoreCredit = increaseCredit (fun c -> c.IsVip) customer1
```

2.3. Open Module2/Tests/Tests.fs, uncomment the tests 2-1, 2-2 and 2-3, save all the files and run the tests by executing Module2/runtests.bat (Win) or Module1/runtests.sh (Mac or Linux) in the Command Prompt/Terminal. Check that the tests pass.

Step 3: Create a vipCondition function

3.1. Create a function called "vipCondition" in the file Module2/Application/Functions.fs that

- Receives a customer as parameter
- Returns true if the customer is VIP or false otherwise

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

3.2. Execute the function in the F# Interactive and test the "increaseCredit" function again but this time using the "vipCondition" function:

```
let customer1WithMoreCredit = increaseCredit vipCondition customer1
```

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

```
let customer1WithMoreCredit = customer1 |> increaseCredit vipCondition
```

3.4. Test "increaseCredit" passing just "vipCondition" in Module2/Application/Try.fsx and check if the result is another function that expects the missing argument (customer):

```
let result = increaseCredit vipCondition
```

You should see the following output:

```
> val result : (Customer -> Customer)
```

3.5. Uncomment tests 2-4 and 2-5, save all the files and run the tests.

Step 4: Create an increaseCreditUsingVip function

4.1. Create a function called "increaseCreditUsingVip" in Module2/Application/Functions.fs by partially applying vipCondition to increaseCredit:

```
let increaseCreditUsingVip = increaseCredit vipCondition
```

4.2. Open Module2/Tests/Tests.fs, uncomment test 2-6, save all the files and run the tests.

Step 5: Create an upgradeCustomer function

5.1. Create a function called "upgradeCustomer" in Module2/Application/Functions.fs that

- Receives a customer as parameter
- Calls getSpendings with the customer and assigns the result to a customerWithSpendings value
- Then it calls tryPromotingToVip passing customerWithSpendings and assigns the result to a promotedCustomer value.
- Then it calls increaseCreditUsingVip with promotedCustomer and assigns the result to an upgradedCustomer value.
- Returns the upgradedCustomer value

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

5.2. Execute "increaseCreditUsingVip" and "upgradeCustomer" in the F# Interactive and test "upgradeCustomer" with customer1 and customer2.

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

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

5.4. Execute the new "upgradeCustomer" in the F# Interactive and test it again with customer1 and customer2.

5.5. Refactor "upgradeCustomer" again to use composition:

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

5.6. Open Module2/Tests/Tests.fs, uncomment tests 2-7 and 2-8, save all the files and run the tests.

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

Step 1: Create an upgradeCustomer function

1.1. Go to the Module3/Application, open Types.fs and create:

- A record called "PersonalDetails" with the following fields:
  - FirstName: string
  - LastName: string
  - DateOfBirth: DateTime
- A discriminated union called "Notifications" with the following cases:
  - NoNotifications
  - ReceiveNotification of receiveDeals: bool * receiveAlerts: bool
- Two units of measure: "AUD" and "USD".
- Then add the following new fields to the Customer (the new types should be declared above "Customer")
  - PersonalDetails: PersonalDetails option
  - Notifications: Notifications
- Finally update the Credit field to use the decimal<USD> type

```
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
}
```

1.2. Highlight all but the "module Types" line and execute it in the F# Interactive (including "open System").

Step 2: Update the increaseCredit function

2.1. Update the “increaseCredit” function to use the USD type in Module3/Application/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> }
```

2.2. Open Module3/Tests/Tests.fs, uncomment the tests 3-1, 3-2 and the customer defined at the top, save all the files and run the tests by executing Module3/runtests.bat (Win) or Module3/runtests.sh (Mac or Linux) in the Command Prompt/Terminal. Check that the tests pass.

Step 3: Create an isAdult function

3.1. Create a function called “isAdult” in Module3/Application/Functions.fs that

- Receives a customer as parameter
- Returns true if the customer is 18 years or older, or false otherwise

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

3.2. Execute “isAdult” in the F# Interactive, open Module3/Application/Try.fsx and send customer1 and customer2 to the F# Interactive, then test isAdult with both customers.

3.3. Open Module3/Tests/Tests.fs, uncomment tests 3-3, 3-4 and 3-5, save all the files and run the tests.

Step 4: Create a getAlert function

4.1. Create a function called “getAlert” in Module3/Application/Functions.fs that

- Receives a customer as parameter
- Returns an option with the string “Alert for customer: #Id” if the customer allowed to receive alerts.

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

4.2. Execute “getAlert” in the F# Interactive and test it with customer1 and customer2.

4.3. Open Module3/Tests/Tests.fs, uncomment tests 3-6 and 3-7, save all the files and run the tests.

Module 4

- Functional lists
- Object-oriented Programming
- Type providers

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

Duration: 20 minutes

Step 1: Create a getSpendings function in the Data module

1.1. Go to the Module4/Application, open Data.fs and create a “getSpendings” function that

- Receives the id of the customer as parameter
- Uses the JsonProvider with the Data.json file (both as schema and data)
- Returns the list of spendings as a sequence

```
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
```

Step 2: Create a getSpendingsByMonth function

2.1. Create a new function called “getSpendingsByMonth” in Module4/Application/Functions.fs right after “tryPromoteToVip” and before “getSpendings” that

- Receives a customer as parameter
- Calls the Data.getSpendings function with the customer’s id and it returns its result

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

2.2. Open Module4/Tests/Tests.fs, uncomment the test 4-1, save all the files and run the tests by executing Module4/runtests.bat (Win) or Module4/runtests.sh (Mac or Linux) in the Command Prompt/Terminal. Check that the test passes.

Step 3: Update the getSpendings function

3.1. Open Module4/Application/Functions.fs and change the implementation of “getSpendings” to:

- Call getSpendingsByMonth passing the customer
- Take the average of the result using List.average
- Return a tuple with the customer and the spendings

```
let getSpendings customer =
    let spending = customer
                   |> getSpendingsByMonth
                   |> List.average
    (customer, spending)
```

3.2. Open Module4/Tests/Tests.fs, uncomment test 4-2, save all the files and run the tests.

Step 4: Create a getCustomers function

4.1. Open Module4/Application/Data.fs and create a “getCustomers” functions that

- Receives zero parameters (unit)
- Uses the CsvProvider with Data.csv (both as schema and data)
- Returns a sequence of customers

```
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 = c.Credit * 1M<USD>
          PersonalDetails = None
          Notifications = NoNotifications
        })
```

Step 5: Create a CustomerService class

5.1. Open Module4/Application/Services.fs and add a “CustomerService” class with

- A GetCustomers method that receives no parameters (unit) and calls the Data.getCustomers function
- An UpgradeCustomers method that receives the id of the customer, it finds the customer using Data.getCustomers and then calls Functions.upgradeCustomers

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

5.2. Open Module4/Tests/Tests.fs, uncomment tests 4-3 and 4-4, save all the files and run the tests.

5.3. Open Module4/Application/Program.fs, uncomment all the code, save all the files and run the application by executing Module4/runtapp.bat (Win) or Module4/runapp.sh (Mac or Linux) in the Command Prompt/Terminal.

5.4. Try the application, upgrade different customers.