F# Workshop

Exercises

Table of Contents

Introduction **2**

Module 1 ..... **5**

Module 2 ..... **8**

Module 3 ..... **10**

Module 4 ..... **12**

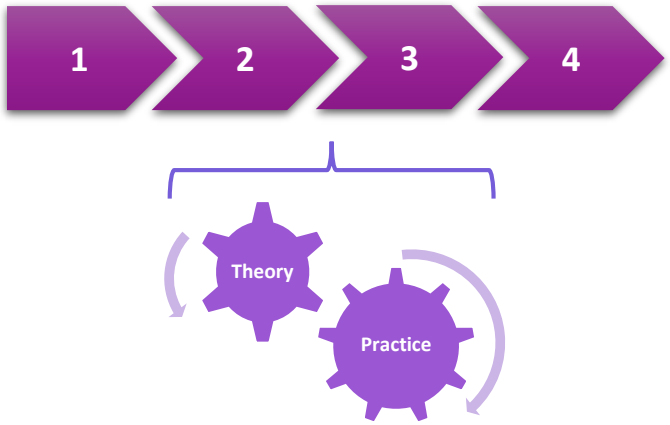
Introduction

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 customer;;
```

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

Duration: 15 minutes

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

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 customer, and execute it in the F# Interactive. Use the following values:

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

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

This should be the result:

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

1.4. 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 purchases as a tuple: (customer, purchases)
- Returns the customer with Vip = true only if the purchases are greather than 100M

```
let tryPromoteToVip (customer, purchases) =
    if purchases > 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 * purchases:decimal -> Customer
```

Note that the function receives a single tuple parameter, as customer and purchases are part the same tuple. In F# commas are used to separate elements of a tuple while spaces are used to separate parameters.

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

- (customer, 101M)

```
let vipCustomer = tryPromoteToVip (customer, 101M)
```

You should see this output:

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

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 getPurchases function

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

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

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

3.2. Execute getPurchases in the F# Interactive. You should see this output:

```
val getPurchases : customer:Customer -> Customer * decimal
```

3.3. Open Module1/Application/Try.fsx and call getPurchases with the customer and execute it in the F# interactive.

```
let purchases = getPurchases customer
```

You should see this output:

```
val purchases : Customer * decimal = ({Id = 1;
                                       IsVip = false;
                                       Credit = 10M;}, 80M)
```

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: 15 minutes

Step 1: Create an increaseCredit function

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

- Receives the condition (function) 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 }
```

1.2. Create a function called "increaseCreditUsingVip" in Module2/Application/Functions.fs by partially applying the "(fun c -> c.IsVip)" lambda as condition to the increaseCredit function:

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

Note that by partially applying the condition you get as result a function that now expects only the customer.

1.3. Execute the function in the F# Interactive and test it in Module2/Application/Try.fsx using the existing customer.

```
let customerWithMoreCredit = increaseCreditUsingVip customer
```

1.4. Open Module2/Tests/Tests.fs, uncomment the tests 2-1, 2-2, 2-3 and 2-4, save all the files and run the tests by executing Module2/runtests.bat (Win) or Module2/runtests.sh (Mac or Linux) in the Command Prompt/Terminal. Check that the tests pass.

Step 2: Create an upgradeCustomer function

2.1. Create a function called “upgradeCustomer” in Module2/Application/Functions.fs that

- Receives a customer as parameter
- Calls getPurchases with the customer and assigns the result to a customerWithPurchases value
- Then it calls tryPromotingToVip passing customerWithPurchases 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 customerWithPurchases = getPurchases customer
    let promotedCustomer = tryPromoteToVip customerWithPurchases
    let upgradedCustomer = increaseCreditUsingVip promotedCustomer
    upgradedCustomer
```

2.2. Execute the function in the F# Interactive and test it in Module2/Application/Try.fsx using the existing customer.

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

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

2.4. Execute the new “upgradeCustomer” in the F# Interactive and test it again Module2/Application/Try.fsx.

2.5. Refactor “upgradeCustomer” again, but this time using composition:

```
let upgradeCustomer = getPurchases >> tryPromoteToVip >> increaseCreditUsingVip
```

Note that the customer parameter needs to be removed when using composition.

2.6. Open Module2/Tests/Tests.fs, uncomment tests 2-5 and 2-6, 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 new types

1.1. Go to the Module3/Application, open Types.fs and create the following types (above the existing Customer type):

- A record called “PersonalDetails” with the following fields:
  - FirstName: string
  - LastName: string
  - DateOfBirth: DateTime
- Two units of measure: “EUR” and “USD”.
- A discriminated union called “Notifications” with the following cases:
  - NoNotifications
  - ReceiveNotification of receiveDeals: bool * receiveAlerts: bool

Then add the following new fields to the 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 EUR
[<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 of age 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. 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 type 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. 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: 15 minutes

Step 1: Refactor the getPurchases function to use the JsonProvider

1.1. Go to the Module4/Application, open Functions.fs and change the "getPurchases" function so that:

- Uses the JsonProvider with the Data.json file (both as schema and data)
- Filters the customer by its id
- Collects the purchases field
- Calculates the purchases' average
- Returns a tuple with the customer and the purchases' average

```
open Types
open FSharp.Data

type Json = JsonProvider<"Data.json">

let getPurchases customer =
    let purchases = Json.Load "Data.json"
                    |> List.ofSeq
                    |> List.filter (fun c -> c.CustomerId = customer.Id)
                    |> List.collect (fun c -> c.PurchasesByMonth)
                    |> List.average
    (customer, purchases)
```

1.2. Open Module4/Tests/Tests.fs, uncomment test 4-1, save all the files and run the tests.

Step 2: Create a CustomerService class

2.1. Open Module4/Application/Services.fs and add a "CustomerService" class with

- An UpgradeCustomers method that receives the id of the customer, it finds the customer using Function.getCustomer and then calls Functions.upgradeCustomer.

```
type CustomerService() =
    member this.UpgradeCustomer id =
        id
        |> Functions.getCustomer
        |> Functions.upgradeCustomer
```

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

2.3. 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.

2.4. Try the application, upgrade different customer ids. The updates are displayed on the screen.