

# F# Workshop

## Exercises Guide



# Introduction

---

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”.

## Pre-requisites

- NET Core SDK
- Visual Studio Code
- Ionide package
- Mono (Mac or Linux only)

## Before we start

---

Make sure you have the pre-requisites installed (see Pre-Requisites section in the previous page).

Please follow these steps to double check your environment is working:

1. Get the source code from <https://github.com/fsharp/fsharp>
2. Open Visual Studio Code
3. Open the root folder (File -> Open Folder)
4. Open the terminal (Terminal -> New Terminal)
5. Run "dotnet test Completed/Module1/Tests"
6. ***Double check it finishes without errors***
7. Open the F# Interactive (View -> Command Palette -> FSI: Start)
8. Write "let a = 1;;" in the terminal window and press enter
9. ***Double check you see "val a : int = 1"***

# Module 1

---

- Bindings
- Functions
- Tuples
- Records

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

Duration: 15-25 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.** Highlight the entire customer type (do not include the “module Types” line) and run View -> Command Palette -> FSI: Send Selection. You should see the following output in the terminal window (F# Interactive):

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


**1.3.** Open Module1/Application/Try.fsx, create a new Customer called customer, and send it in the F# Interactive (View -> Command Palette -> FSI: Send Selection). Use the following values:

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

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

Note that you only need to send that line to the F# interactive, ignore the rest of the content of the file.

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 by selecting the lines 8 to 12 and running View -> Command Palette -> Remove Line Comment. Save all the files (File -> Save All), go to the Terminal, select the command terminal (bash, cmd or powershell) and run “dotnet test Module1/Tests”.


## Step 2: Create a tryPromoteToVip function

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

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

```
let tryPromoteToVip purchases =  
    let customer, amount = purchases  
    if amount > 100M then { customer with IsVip = true }  
    else customer
```

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


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

Note that the function receives a single tuple parameter containing the customer and purchases. In F# commas separate elements of a tuple while spaces separate parameters.

2.3. Save all the files and open Module1/Application/Try.fsx, invoke the tryPromoteToVip function with the values “(customer, 101M)” and assign the result to a value called vipCustomer. Then send it to the F# Interactive.

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

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 “dotnet test Module1/Tests” in the command terminal.


### Step 3: Create a `getPurchases` function

**3.1.** Add a function called “`getPurchases`” to `Module1/Application/Functions.fs` that

- Receives a customer as a parameter
- Returns a tuple with the customer and his/her 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.** Send `getPurchases` to 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 calculatedPurchases = getPurchases customer
```

You should see this output:

```
 val calculatedPurchases : 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 “dotnet test Module1/Tests” in the command terminal.

## Module 2

- High order functions
- Pipelining
- Partial application
- Composition

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

Duration: 15-20 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 as parameter.

1.3. Send both functions (increaseCredit and increaseCreditUsingVip) to the F# Interactive and test the latter in Module2/Application/Try.fsx using the existing customer.

```
let customerWithMoreCredit = increaseCreditUsingVip customer
```

You should see this output:

```
 val customerWithMoreCredit : Customer = {Id = 1;  
                                           IsVip = false;  
                                           Credit = 60M;}
```

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 “dotnet test Module2/Tests” in the command terminal.

## 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 tryPromoteToVip 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. Send the function to the F# Interactive and test it in Module2/Application/Try.fsx using the existing customer and assigning the result to an upgradedCustomer value.

2.3. Refactor the “upgradeCustomer” function to use the pipelining operator:

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

2.4. Send the new “upgradeCustomer” to the F# Interactive and test it again in 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 “dotnet test Module2/Tests” in the command terminal.



## Module 3

- Options
- Pattern matching
- Discriminated unions

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

Duration: 15-20 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
- 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
}

type Notifications =
    | NoNotifications
    | ReceiveNotifications of receiveDeals: bool * receiveAlerts: bool

type Customer = {
    Id: int
    IsVip: bool
    PersonalDetails: PersonalDetails option
    Notifications: Notifications
}
```

1.2. Highlight all but the “module Types” line and send it to the F# Interactive (including “open System”).

## Step 2: Update the tryPromoteToVip function

2.1. Update the “tryPromoteToVip” function to use the “amount” of the purchases in Module3/Application/Functions.fs. Use the “amount” to update the customer status to VIP is the amount in > then 100.

```
let tryPromoteToVip purchases =  
    let customer, amount = purchases  
    if amount > 100M then { customer with IsVip = true }  
    else customer
```

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 “dotnet test Module3/Tests” in the command terminal.

## 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 false if the PersonalDetails are not defined (None)
- 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 “dotnet test Module3/Tests” in the command terminal.

## 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 “Alert for customer [Id]” if the customer allowed alerts or returns an empty string otherwise.

```
let getAlert customer =  
    match customer.Notifications with  
    | ReceiveNotifications(receiveAlerts = true) ->  
        sprintf "Alert for customer %i" customer.Id  
    | _ -> ""
```

4.2. Open Module3/Tests/Tests.fs, uncomment tests 3-6 and 3-7, save all the files and run “dotnet test Module3/Tests” in the command terminal.

## Module 4

- Functional lists
- Units of measure
- Object-oriented
- Object Expression

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

Duration: 15-20 minutes

### Step 1: Add Unit of Measure

1.1. Go to the Module4/Application, open Types.fs and add/create the following types:

- Two units of measure: “EUR” and “USD”.
- Update the Credit field to use the decimal<USD> type

Then add the following new fields to the Customer:

- PersonalDetails: PersonalDetails option
- Notifications: Notifications

```
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 send it to the F# Interactive (including “open System”).

### Step 2: Update the increaseCredit function

2.1. Update the “increaseCredit” function to use the USD type in Module4/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. Update the “isAdult” function to use the USD type in Module4/Application/Functions.fs:

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

2.3. Update the “getAlert” function to use the USD type in Module4/Application/Functions.fs:

```
let getAlert customer =
    match customer.Notifications with
    | ReceiveNotifications(receiveAlerts = true) ->
        sprintf "Alert for customer %i" customer.Id
    | _ -> ""
```

2.4. Open Module4/Tests/Tests.fs, uncomment the tests 4-1, 4-2 and the customer defined at the top, save all the files and run “dotnet test Module3/Tests” in the command terminal.

## Step 1: Refactor the getPurchases function to use the System.IO.File

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

- Uses the File.ReadAllLines (System.IO) with the Data.txt file (comma separated)
- Parse the lines of the text file to instantiate a new type ‘PurchaseHistory’ (see Types.fs)
  - Use the String ‘Split’
  - In each line, the first cell is the ‘CustomerId’, the other following cells are the purchases
- Filters the customer by his/her id
- Collects the PurchasesByMonth field
- Calculates the purchases’ average
- Returns a tuple with the customer and the purchases’ average

```
let [<Literal>] dataPath = __SOURCE_DIRECTORY__ + "/Data/Customers.txt"

let getPurchases customer =
    let purchases =
        File.ReadAllLines(dataPath)
        |> Seq.map(fun line ->
            let cells = line.Split(',', StringSplitOptions.RemoveEmptyEntries)
            {
                PurchaseHistory.CustomerId = int cells.[0]
                PurchaseHistory.PurchasesByMonth = cells.[1..]
            }
            |> Seq.map decimal |> Seq.toList
        })
    |> Seq.filter (fun c -> c.CustomerId = customer.Id)
    |> Seq.collect (fun c -> c.PurchasesByMonth)
    |> Seq.average
    (customer, purchases)
```

1.2. Open Module4/Tests/Tests.fs, uncomment test 4-1, save all the files and run “dotnet test Module4/Tests” in the command terminal.

## Step 2: Create a CustomerService class with an UpgradeCustomer method

2.1. Open Module4/Application/Services.fs and add an interface “ICustomerService with an UpgradeCustomers method that receives an id and return a Customer. Then add another method “GetCustomerInfo” that receives a customer and return a string.

```
type ICustomerService =  
    abstract UpgradeCustomer : int -> Types.Customer  
    abstract GetCustomerInfo : Types.Customer -> string
```

Successively, implement a class ‘CustomerService’ that inherits and satisfies the previously defined interface ‘ICustomerService’.

Here is the implementation details.

The method ‘UpgradeCustomer’ should:

- Receive the id of the customer
- Find the customer using Function.getCustomer
- And then call Functions.upgradeCustomer

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

The method called “GetCustomerInfo” should:

- Receive a customer as parameter
- Calculate whether the customer is adult or not using the Functions.isAdult function
- Get the alert using the Functions.getAlert function
- Return a string with the format "Id: [Id], IsVip: [IsVip], Credit: [Credit], IsAdult: [IsAdult], Alert: [Alert]"

```
type CustomerService() =  
    ...  
    member this.GetCustomerInfo customer =  
        let isAdult = Functions.isAdult customer  
        let alert = Functions.getAlert customer  
        sprintf "Id: %i, IsVip: %b, Credit: %.2f, IsAdult: %b, Alert: %s"  
            customer.Id customer.IsVip customer.Credit isAdult alert
```

Here is the implementation of the ‘CustomerService’ class:

```

type CustomerService() =

    interface ICustomerService with
        member this.UpgradeCustomer id =
            id
            |> Functions.getCustomer
            |> Functions.upgradeCustomer

        member this.GetCustomerInfo customer =
            let isAdult = Functions.isAdult customer
            let alert = Functions.getAlert customer
            sprintf "Id: %i, IsVip: %b, Credit: %.2f, IsAdult: %b, Alert: %s"
                customer.Id customer.IsVip customer.Credit isAdult alert

```

3.2. Open Module4/Tests/Tests.fs, uncomment test 4-3, save all the files and run “dotnet test Module4/Tests” in the command terminal.

### Step 3: Create a CustomerService using Object-Expression

3.3. Open the ‘Program.fs’ file, remove the instantiation of the ‘CustomerService’. Replace the ‘service’ instance with an inline ObjectExpression that implements the interface ‘ICustomerService’.

```

let service =
    { new ICustomerService with
        member this.UpgradeCustomer id =
            id
            |> Functions.getCustomer
            |> Functions.upgradeCustomer

        member this.GetCustomerInfo customer =
            let isAdult = Functions.isAdult customer
            let alert = Functions.getAlert customer
            sprintf "Id: %i, IsVip: %b, Credit: %.2f, IsAdult: %b, Alert: %s"
                customer.Id customer.IsVip customer.Credit isAdult alert }

```

### Step 4: Run the application

4.1. Open Module4/Application/Program.fs, uncomment all the code, save all the files and run “dotnet run -p Module4/Application” in the command terminal.

4.2. Try the application, upgrade different customer ids. You should see the following output:

```

Id to upgrade [1-4]: 2

Customer to upgrade:
Id: 2, IsVip: false, Credit: 10.00, IsAdult: false, Alert: Alert for customer 2

Upgrading customer...

Customer upgraded:
Id: 2, IsVip: true, Credit: 110.00, IsAdult: false, Alert: Alert for customer 2

Press any key to try again or 'q' to quit

```

Note that we are not saving the updates, they are just displayed on the screen. Trying the same customer id multiple times will generate the same output.

## Module 5

---

- Recursion
- Active Patterns

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

Duration: 20-25 minutes

### Step 1: Implement a Tail Recursive function

- 1.1. Go to the Module5/Recursion.fsx and add/create a tail recursion version of the existing “addOne” function. The optimized functions should not throw an exception when running using a list of 40000 items.

### Step 2: Implement a “FizzBuzz” game using Active Patterns

- 1.2. Go to the Module5/FizzBuzz.fsx and add/create a version of the “fiizBuzz” game using Active Patterns.

## Module 6

---

- DSL
- Concurrency
- Async programming
- Agent (MailBoxProcessor)

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

Duration: 30-35 minutes

### Step 1: Building a DSL in F#

(Run Demo Calculator)

Let’s implement a small DSL to order a coffee. Open the file `DSL.fsx`.

Every morning on your way to the office, you pull your car up to your favorite coffee shop

for a Grande Skinny Cinnamon Dolce Latte with whip. The barista always serves you exactly what you order.

The barista can do this because you placed your order using precise language that he/she understands. You don’t have to explain the meaning of every term that you utter, though to others what you say might be incomprehensible

- 5.1. Implement a set of Discriminated Unions (DU) to define the language to use to order a coffee.

First the `size` of the coffee, can be either `Tall` or `Grande` or `Venti`

```
type size = Tall | Grande | Venti
```

Now, implement the DUs to define both the type of drink (`Latte`, `Cappuccino`, `Mocha`, `Americano`) and the extras (`Shot` and `Syrup`).

```
type drink = Latte | Cappuccino | Mocha | Americano
type extra = Shot | Syrup
```

**5.2.** Implement a Record Type that defines the Cup of coffee, with the properties of `Size`, `Drink` and the `Extras` that expose the DUs previously defined.

```
type Cup = { Size:size; Drink:drink; Extras:extra list }
```

In addition, add two `static` member to the `Cup` Record Type.

One static method `Of` that take a `size` and a `drink` as an argument, and returns a `Cup`. This is a helper function that is quite useful for starting the DSL.

```
static member (+) (cup:Cup,extra:extra) =
    { cup with Extras = extra :: cup.Extras }
```

Next, add another static member that defines the plus infix operator `+`. This operator helps to add `extra` to a given `Cup`. Thus, the two arguments passed are a `Cup` and a `extra` type.

```
static member Of size drink =
    { Size=size; Drink=drink; Extras=[] }
```

Now, for testing, using this DSL you should be able to order a `Grande Latte with an extra shot`

```
let coffe = Cup.Of Grande Latte + Shot
```

**5.3.** Uncomment the function `Price` and calculate the cost of your favorite coffee drink.

## Step 1: Implement a parallel WebCrawler

**1.1.** Create an Agent that prints a given message. fields:

```
let printerAgent =
    Agent<Msg<string, string>>.Start((fun inbox -> async {
        while true do
            let! msg = inbox.Receive()
            match msg with
            | Item(t) -> printfn "%s" t
            | Mailbox(agent) -> failwith "no implemented"}), cancellationToken = cts.Token)
```

**1.2.** Create a “parallelAgent” worker based on the MailboxProcessor. The idea is to have an Agent that handles, computes and distributes the messages in a Round-Robin fashion between a set of Agent children.

```
let parallelAgent n f =
    let agents = Array.init n (fun _ ->
        Agent<Msg<'a, 'b>>.Start(f, cancellationToken = cts.Token))

    let token = cts.Token

    let agent = new Agent<Msg<'a, 'b>>((fun inbox ->
        let rec loop index = async {
            let! msg = inbox.Receive()
            match msg with
            | Msg.Item(item) ->
                agents.[index].Post (Item item)
                return! loop ((index + 1) % n)
            | Mailbox(agent) ->
                agents |> Seq.iter(fun a -> a.Post (Mailbox agent))
                return! loop ((index + 1) % n)
        })
    }
```



```

        loop 0), cancellationToken = token)

token.Register(fun () -> agents |> Seq.iter(fun agent ->
                                         (agent :> IDisposable).Dispose())) |> ignore
agent.Start()
agent

```

### 1.3. Complete the Agent case “Item(url)” in step (3)

```

if urls |> Set.contains url |> not then
    let! content = downloadContent url
    content |> Option.iter(fun c ->
        for agent in agents do
            agent.Post (Item(c)))
    let urls' = (urls |> Set.add url)
    match limit with
    | Some l when urls' |> Seq.length >= l -> cts.Cancel()
    | _ -> return! loop urls' agents
else return! loop urls agents

```

### 1.4. Create a broadcast agent, which sends a message to all the sub-agents registered as subscribers. The message is sent (broadcast) to all the agent subscribed. The agent registration is done using the “Mailbox(agent)”. The agents list (subscribers) is kept as state in the agent rec-loop (agents: Agent<\_> list)

```

let broadcastAgent () =
    parallelAgent parallelism (fun inbox ->
        let rec loop (agents : Agent<_> list) = async {
            let! msg = inbox.Receive()
            match msg with
            | Item(item) ->
                for agent in agents do
                    agent.Post(Item(item))
                return! loop agents
            | Mailbox(agent) -> return! loop (agent::agents)
        }
        loop [])

```

### 1.5. Implement a “link” agent parser. Follow step (5).

### 1.6. Run the “Program.fs” to the project.

**i** let agent = new ParallelWebCrawler.WebCrawler()  
 agent.Submit "https://www.google.com"

Console.ReadLine() |> ignore