F# Introduction Workshop

by Jorge Fioranelli @jorgefioranelli

V1.0

Objectives

- Understand the basic core principles behind FP
- Understand the F# syntax
- Understand the F# structures
- Get motivation to practice and master F#

Disclaimer

- Your brain will hurt
- You will need to keep practicing
- This is just an introduction
- This is not a “C# vs F#” session
- The code is not production-ready

Materials

- Exercises Document
- Exercises source code
- F# Cheatsheet

fsharpworkshop.com

github.com/jorgef/fsharpworkshop

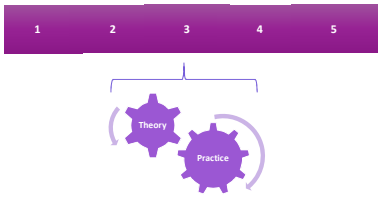
Minimum Requirements

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

Nuget Packages

- XUnit
- Unquote
- SqlProvider (TypeProvider) [alpha]
- F# Data

Modules



Module 1

Bindings | Functions | Tuples | Records

Module 2

High order functions | Pipelining | Partial application | Composition

Module 3

Options | Pattern matching | Discriminated unions | Units of measure

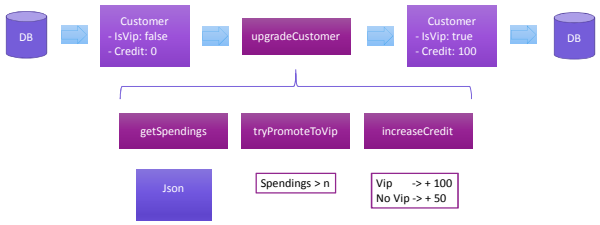
Module 4

Functional lists | Recursion | List module

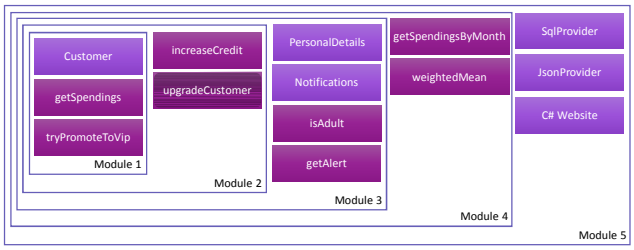
Module 5

Object Oriented Programming | Type providers

Exercise



Exercise



Module 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

F# is a strongly-typed,
functional-first language
for writing simple code
to solve complex problems.

Imperative vs Functional

Conventions

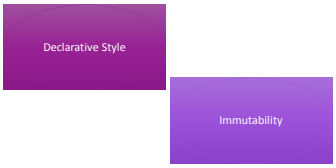
C#

```
var number = 1;
```

F#

```
let number = 1
```

Functional Core Concepts



Declarative Style

Imperative →

```
var vipCustomers = new List<Customer>();
foreach (var customer in customers)
{
    if (customer.IsVip)
        vipCustomers.Add(customer);
}
```

Declarative →

```
var vipCustomers = customers.Where(c => c.IsVip);
```

Immutability

```
var x = 1;   ≠   let x = 1
```

~~let mutable x = 1~~
~~x <- 2~~

~~x = x + 1~~

```
let y = x + 1
```

Functions

```
int Sum(int num1, int num2)
{
   var result = num1 + num2;
   return result;
}

int Sum(int num1, int num2)
{
   return num1 + num2;
}

int Sum(int num1, int num2)
```

in out

Func<int,int,int>

name parameters (type inference)

```
let sum num1 num2 =
   let result = num1 + num2
   result
```

body

return

```
let sum num1 num2 =
   num1 + num2

let sum num1 num2 = num1 + num2

sum : num1:int -> num2:int -> int
```

in out

int -> int -> int

Pure Functions and Side Effects

```
public int Sum(int a, int b)
{
   return a + b;
}
```

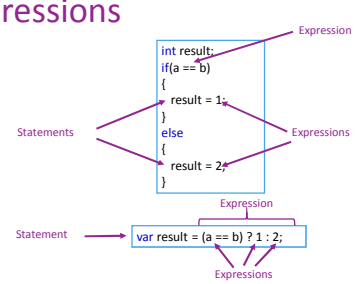
```
private int accumulator;

public int Sum(int a, int b)
{
   accumulator++;
   return a + b;
}
```

Expressions

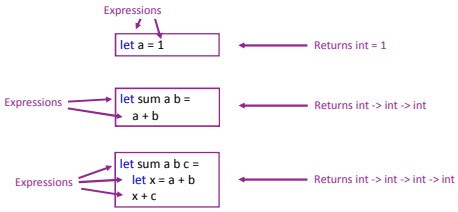
Expression `a == b` ← Returns a Boolean

Statement `var a = 1;` ← Doesn't return anything

Expressions



Bindings



Tuples

```
Tuple<int, int> Divide(int dividend, int divisor)
{
   var quotient = dividend / divisor;
   var remainder = dividend % divisor;
   return new Tuple<int, int>(quotient, remainder);
}
```

```
let divide dividend divisor =
   let quotient = dividend / divisor
   let remainder = dividend % divisor
   (quotient, remainder)
```

```
var result = Divide(10, 3);
var quotient = result.Item1;
var remainder = result.Item2;
```

```
let quotient, remainder = divide 10 3
```

```
let success, value = Int32.TryParse("42")
```

Records

```
public class DivisionResult
{
   public int Quotient { get; set; }
   public int Remainder { get; set; }
}
```

```
type DivisionResult =
{ Quotient    : int
  Remainder : int }
```

```
public class DivisionResult
{
   private readonly int quotient;
   private readonly int remainder;
   public DivisionResult(int quotient, int remainder)
   {
      this.quotient = quotient;
      this.remainder = remainder;
   }
   public int Quotient
   {
      get { return quotient; }
   }
   public int Remainder
   {
      get { return remainder; }
   }
}
```

Records

```
type DivisionResult =
{ Quotient    : int
  Remainder : int }
```

```
let result = { Quotient = 3; Remainder = 1 }
```

```
let result = { Quotient = 3 }
```

Error: No assignment given for field 'Remainder' of type

```
let newResult = { Quotient = result.Quotient; Remainder = 0 }
```

```
let newResult = { result with Remainder = 0 }
```

```
let result1 = { Quotient = 3; Remainder = 1 }
let result2 = { Quotient = 3; Remainder = 1 }
result1 = result2 // true
```

Structural Equality
Reference Types

Immutable and Structural Equality

```
var message1 = "hello John Doe";
var message2 = "hello John Doe";
```

```
var result = message1 == message2; // true
```

```
var message3 = message1.Replace("hello", "hi");
```

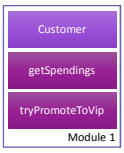
F# in Visual Studio

- F# Interactive
- Scripts vs Source Files
- Order matters
- No folders

Demo 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

Exercise 1



Exercise 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

Review

- How do you return a value in a function?
- How many parameters has tryPromoteToVip?
- Can you explain this type? string -> int -> object
- How do you change a Record?
- Can you explain what is the “it” word in some of the outputs?

Module 2

HIGH ORDER FUNCTIONS | PIPELINING | PARTIAL APPLICATION | COMPOSITION

High Order Functions

```
public int Sum(int a, int b)
{
    return a + b;
}
```

```
public int Execute(int a, int b, Func<int,int,int>operation)
{
    return operation(a, b);
}
```

```
var result = Execute(1, 2, (a,b) => a + b);
```

```
var result = Execute(1, 2, (a,b) => a * b);
```

```
var result = Execute(1, 2, Sum);
```

High Order Functions

High Order Functions

```
var productNames = products
    .Where(p => p.Category == productCategory)
    .Select(p =>p.Name);
```

```
public Func<int,int,int> GetOperation(Type operationType)
{
   if (operationType == Type.Sum)
      return (a, b) => a + b;
   else
      return (a, b) => a * b;
}

var operation = GetOperation(type);
```

High Order Functions

```
let sum a b = a + b
```

```
let execute a b op = op a b
```

```
let getOperation type =
   if type = OperationType.Sum then fun a b -> a + b
   else fun a b -> a * b
```

```
let getOperation type =
   if type = OperationType.Sum then (+)
   else (*)
```

Extension Methods in C#

```
public List<int> Filter(List<int> list, Func<int,bool>condition)
```

```
public static List<int> Filter(this List<int> list, Func<int,bool>condition)
```

```
var filteredNumbers = Filter(numbers, n => n > 1);
```

```
var filteredNumbers = numbers.Filter(  n => n > 1);
```

```
var filteredNumbers = numbers
                      .Filter(n => n > 1)
                      .Filter(n => n < 3);
```

Pipelining Operator

```
public List<int> Filter(List<int> items, Func<int,bool>condition)
```

```
let filter condition items = // ...
```

```
let filteredNumbers = filter (fun n -> n > 1) numbers
```

```
let filteredNumbers = numbers |> filter (fun n -> n > 1)
```

```
let filteredNumbers = numbers
                      |> filter (fun n -> n > 1)
                      |> filter (fun n -> n < 3)
```

Partial Application

```
let sum a b = a + b
```

```
let result = sum 1 2
```
← Returns int = 3

```
let result = sum 1
```
← Returns int -> int

```
let addOne = sum 1
```
← Returns int -> int

```
let result = addOne 2
```
← Returns int = 3

```
let result = addOne 3
```
← Returns int = 4

Composition

```
let addOne a = a + 1
```

```
let addTwo a = a + 2
```

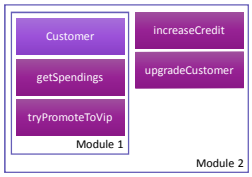
```
let addThree = addOne >> addTwo
```

```
let result = addThree 1
```
← Returns int = 4

Demo 2

HIGH ORDER FUNCTIONS | PIPELINING | PARTIAL APPLICATION | COMPOSITION

Exercise 2



Exercise 2

HIGH ORDER FUNCTIONS | PIPELINING | PARTIAL APPLICATION | COMPOSITION

Review

- What keyword do you use for lambda expressions?
- What happens if the function I need is defined after the caller?
- What happens when a function is called without all its parameters?
- Why |> is better than the Extension Methods?

Module 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

NullReferenceExceptions

```
var customer = GetCustomerById(42);
var isAdult = customer.Age >= 18;
```

```
public Customer GetCustomerById(int id)
```

NullReferenceException

```
if (customer == null)
   throw new Exception("Not found");
var isAdult = customer.Age >= 18;
```

```
if (customer == null)
   // Try something different
else
   var isAdult = customer.Age >= 18;
```

NullReferenceExceptions

```
var age = GetCustomerAgeById(42);
var isAdult = age >= 18;
```

```
public int GetCustomerAgeById(int id)
```

Hint: Possible Null

```
var isAdult = age.Value >= 18;
```

```
public int? GetCustomerAgeById(int id)
```

```
if (!age.HasValue)
   // Try something different
else
   var isAdult = age.Value >= 18;
```

Options

int

Nullable<int>

Customer

Nullable<Customer>

int

Option<int> → None, Some of int

Customer

Option<Customer> → None, Some of Customer

Options

```
let addOne x = x + 1
```

In: int → Out: int

int option -> int option

2 → addOne → 3

Some 2 → addOne → Some 3

None → addOne → None

Options

```
let addOne x = x + 1
```
← int -> int

```
let addOne (x: int option) =
   if x= None then 0
   else x.Value + 1
```
← int option -> int

```
let addOne x =
   if x = None then 0
   else x.Value + 1
```
← int option -> int

```
let addOne x =
   if x = None then None
   else Some (x.Value + 1)
```
← int option -> int option

Pattern Matching

```
let addOne x =
   if x = None then None
   else Some (x.Value + 1)
```

```
let addOne x =
   match x with
   | None -> None
   | Some n -> Some (n + 1)
```

Discriminated Unions

```
public abstract class DivisionResult
{
}
public class DivisionSuccess : DivisionResult
{
   public int Quotient { get; set; }
   public int Remainder { get; set; }
}
public class DivisionError : DivisionResult
{
   public string ErrorMessage { get; set; }
}
```

```
type DivisionResult =
| DivisionSuccess of quotient : int * remainder : int
| DivisionError of message : string
```

Discriminated Unions

```
let result = divide 2 0
match result with
| DivisionSuccess (quotient, remainder) ->
    printfn "Quotient:%i Remainder:%i" quotient remainder
| DivisionError message ->
    printfn "Error: %s" message
```

```
let divide dividend divisor =
   match divisor with
    |0 -> DivisionError (message = "Divide by zero")
    |_ -> DivisionSuccess (quotient = dividend / divisor,
                           remainder = dividend % divisor)
```

Units of Measure

```
let distanceInMts = 11580.0
let distanceInKms = 87.34
let totalDistance = distanceInMts + distanceInKms
```

← 11667.34

```
[<Measure>] type m
[<Measure>] type km

let distanceInMts = 11580.0<m>
let distanceInKms = 87.34<km>
let totalDistance = distanceInMts + distanceInKms
```

↑

Error: The unit of measure 'm' does not match the unit of measure 'km'

Units of Measure

```
[<Measure>] type km
[<Measure>] type h

let time = 2.4<h>
let distance = 87.34<km>
let speed = distance / time
```

← 36.39<km/h>

```
[<Measure>] type m

let width = 2<m>
let height = 3<m>
let surface = width * height
```

← 6<m^2>

Units of Measure

```
let distanceInMts = 11580.0<m>
let distanceInKms = 87.34<km>
let totalDistance = distanceInMts + distanceInKms
```

Error: The unit of measure 'm' does not match the unit of measure 'km'

```
let mts2Kms (m : float<m>) = m / 1.0<m> / 1000.0 * 1.0<km>
```

float<m> -> float<km>

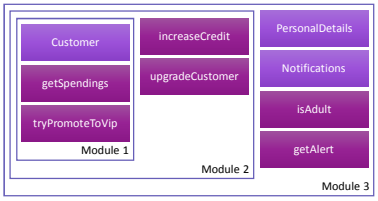
```
let totalDistance = (mts2Kms distanceInMts) + distanceInKms
```

← 98.920<km>

Demo 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

Exercise



Exercise 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

Review

- How do you convert two units of measure?
- Why do we use "%i" in the sprintf function?
- Why do we use "_"?

Module 4

FUNCTIONAL LISTS | RECURSION | LIST MODULE

Functional Lists

```
var numbers = new List<int>{2, 3, 4};
numbers.Insert(0, 1);
```

```
numbers.AddRange(new List<int>{5, 6});
```

```
var ns = Enumerable.Range(1, 1000).ToList();
```

```
var empty = new List<int>();
```

```
let numbers = [2; 3; 4]
let newNumbers = 1 :: numbers
```

```
let twoLists = numbers @ [5; 6]
```

```
let ns =[1 .. 1000]
```

```
let empty = []
```

```
let odds =[1 .. 2 .. 1000]
```

```
let oddsWithZero =[ yield 0
                    yield! odds ]
```

```
let gen = [ for n in numbers do
              if n%3 = 0 then
                yield n * n ]
```

Lists vs Arrays vs Sequences

List

```
let myList = [1; 2]
```

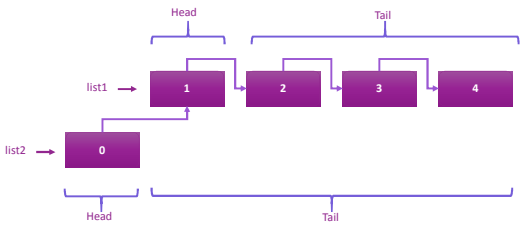
Array

```
let myArray = [|1; 2|]
```

Seq

```
let mySeq = seq { yield 1; yield 2 }
```

Functional Lists



Processing Lists

```
let numbers = [1..4]
let mutable result = [] : int list
for n in numbers do
 if n % 2 = 0 then result <- n :: result
```

```
let rec even ls =
 match ls with
 |[] -> []
 |head :: tail when head % 2 = 0 ->head :: even tail
 |_ :: tail  else even tail
```

Recursive Function

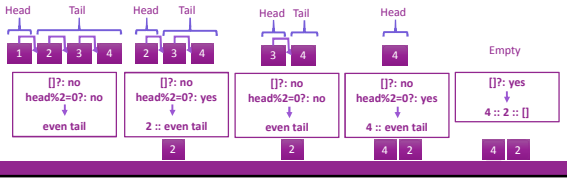
Empty List (end)

Non Empty List

Decompose List

Recursion

```
let rec even ls =
 match ls with
 |[] -> []
 |head :: tail when head % 2 = 0 -> head :: even tail
 |_ :: tail  else even tail
```



Tail Recursion

```
let rec even ls =
 match ls with
 |[] -> []
 |head :: tail when head % 2 = 0 ->head :: even tail
 |_ :: tail  else even tail
```

```
let rec even ls acc =
 match ls with
 |[] -> acc
 |head :: tail when head % 2 = 0 -> even tail (head :: acc)
 |_ :: tail  else even tail acc
```

List Module

List.filter	.Where
List.map	.Select
List.fold	.Aggregate
List.find	.First
List.tryFind	.FirstOrDefault
List.forall	.All
List.exist	.Any
List.partition	-
List.zip	.Zip
List.rev	.Reverse
List.collect	.SelectMany
List.choose	-
List.pick	-
List.toSeq	.AsEnumerable
List.ofSeq	.ToList

Complete list: http://msdn.microsoft.com/en-us/library/ee353738.aspx

List Module

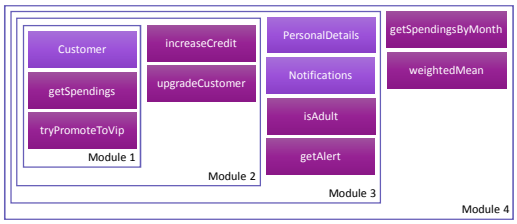
```
let vipNames = customers
               |> List.filter (fun c -> c.IsVip)
               |> List.map (fun c -> c.Name)
```

```
var vipNames = customers
               .Where(c => c.IsVip)
               .Select(c => c.Name);
```

Demo 4

FUNCTIONAL LISTS | RECURSION | LIST MODULE

Exercise 4



Exercise 4

FUNCTIONAL LISTS | RECURSION | LIST MODULE

Review

- What does List.zip do?
- Why do we use an accumulator in the recursiveWeightedMean function?
- Why to we wrap recursiveWeightedMean inside recursiveWeighted?

Module 5

OBJECT ORIENTED PROGRAMMING | TYPE PROVIDERS

Classes – Immutable Properties

```
public class MyClass
{
   private readonly int myFiled;
   public MyClass(int myParam)
   {
      myField = myParam;
   }
   public int MyProperty
   {
      get { return myField; }
   }
}
```

```
type MyClass(myField: int) =
   member this.MyProperty = myField
```

Classes – Mutable Properties

```
public class MyClass
{
   public MyClass(int myParam)
   {
      MyProperty = myParam;
   }
   public int MyProperty { get; set; }
}
```

```
type MyClass(myField: int) =
   let mutable myMutableField = myField
   member this.MyProperty
      with get () = myMutableField
      and set(value) = myMutableField <- value
```

Classes – Public Methods

```
public class MyClass
{
   private readonly int myFiled;
   public MyClass(int myParam)
   {
      myFiled = myParam;
   }
   public int MyMethod(int methodParam)
   {
      return myFiled + methodParam;
   }
}
```

```
type MyClass(myField int) =
   member this.MyMethod methodParam =
      myField + methodParam
```

Classes – Private Methods

```
public class MyClass
{
   public int MyMethod(int methodParam)
   {
      return myPrivateMethod(methodParam);
   }
   private int MyPrivateMethod(int methodParam)
   {
      return methodParam + 1;
   }
}
```

```
type MyClass() =
   let myPrivateFun funParam =
      funParam + 1
   member this.MyMethod methodParam =
      myPrivateFun methodParam
```

Classes – Inheritance

```
public abstract class MyBaseClass
{
   public abstract int MyMethod(int methodParam);
}

public class MyClass : MyBaseClass
{
   public override int MyMethod(int methodParam);
   {
      return methodParam + 1;
   }
}
```

```
[<AbstractClass>]
type MyBaseClass() =
   abstract member this.MyMethod: int -> int

type MyClass() =
   inherits MyBaseClass ()
   override this.MyMethod methodParam =
      methodParam + 1
```

Classes – Interfaces

```
public interface IMyInterface
{
   int MyMethod(int methodParam);
}

public class MyClass : IMyInterface
{
   public int MyMethod(int methodParam);
   {
      return methodParam + 1;
   }
}
```

```
type IMyInterface =
   abstract member this.MyMethod: int -> int

type MyClass() =
   interface IMyInterface with
      member this.MyMethod methodParam =
         methodParam + 1
```

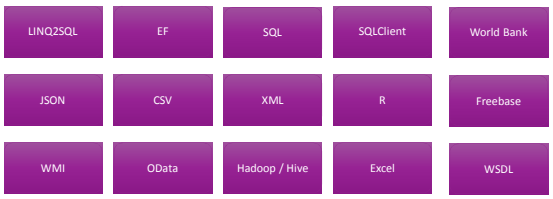
Classes – Object Expressions

```
public interface IMyInterface
{
   int MyMethod(int methodParam);
}

public class MyClass : IMyInterface
{
   public int MyMethod(int methodParam);
   {
      return methodParam + 1;
   }
}
```

```
type IMyInterface =
   abstract member this.MyMethod: int -> int

let myInstance =
   { new IMyInterface with
      member this.MyMethod methodParam =
         methodParam + 1 }
```

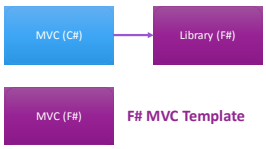
Type Providers

Type Providers



And many more!

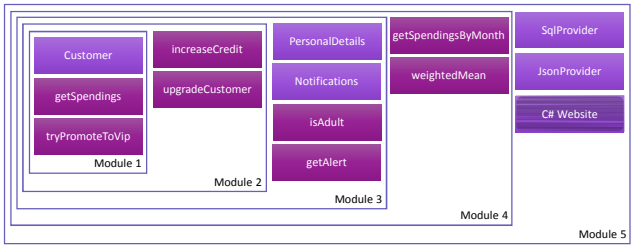
ASP.NET MVC



F# MVC Template

All options: http://fsharp.org/guides/web/

Exercise 5



Exercise 5

OBJECT ORIENTED PROGRAMMING | TYPE PROVIDERS

Thank you