# F# Workshop

BY JORGE FIORANELLI - @JORGEFIORANELLI

# Objectives

> Understand the basic core principles behind FP

> Understand the F# syntax and structures

> Get motivation to practice and master F#

# Materials

> Exercises Guide

> Exercises Source Code

fsharpworkshop.com

github.com/jorgef/fsharpworkshop

# Pre-requisites

## > Windows

> Visual Studio 2015 Community or

> Xamarin Studio or

> Atom + F# Compiler + Ionide package or
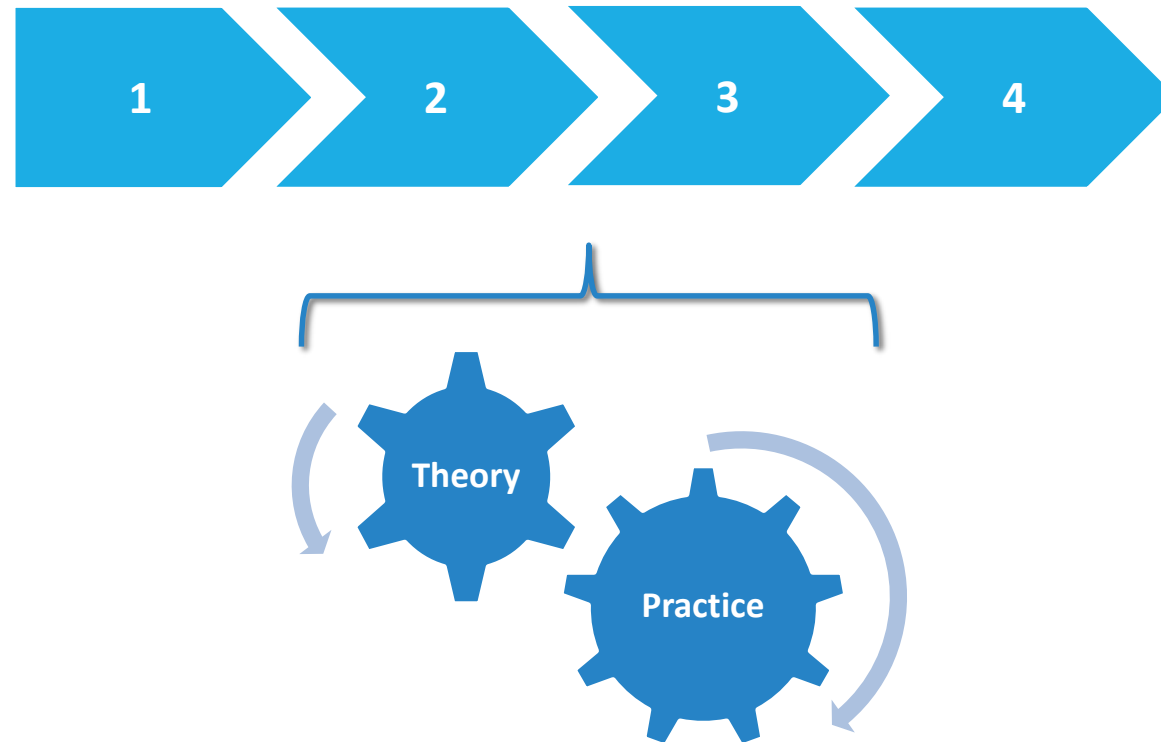
> Visual Studio Code + F# Compiler + Ionide package

## > Linux

> Atom + Mono + Ionide package or

> Visual Studio Code + Mono + Ionide package

## > Mac

> Xamarin Studio or

> Atom + Mono + Ionide package or

> Visual Studio Code + Mono + Ionide package

# Modules

# Agenda

## 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 | Object-oriented programming | Type providers

# Module 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

F# is a mature, open source, cross-platform, functional-first programming language.
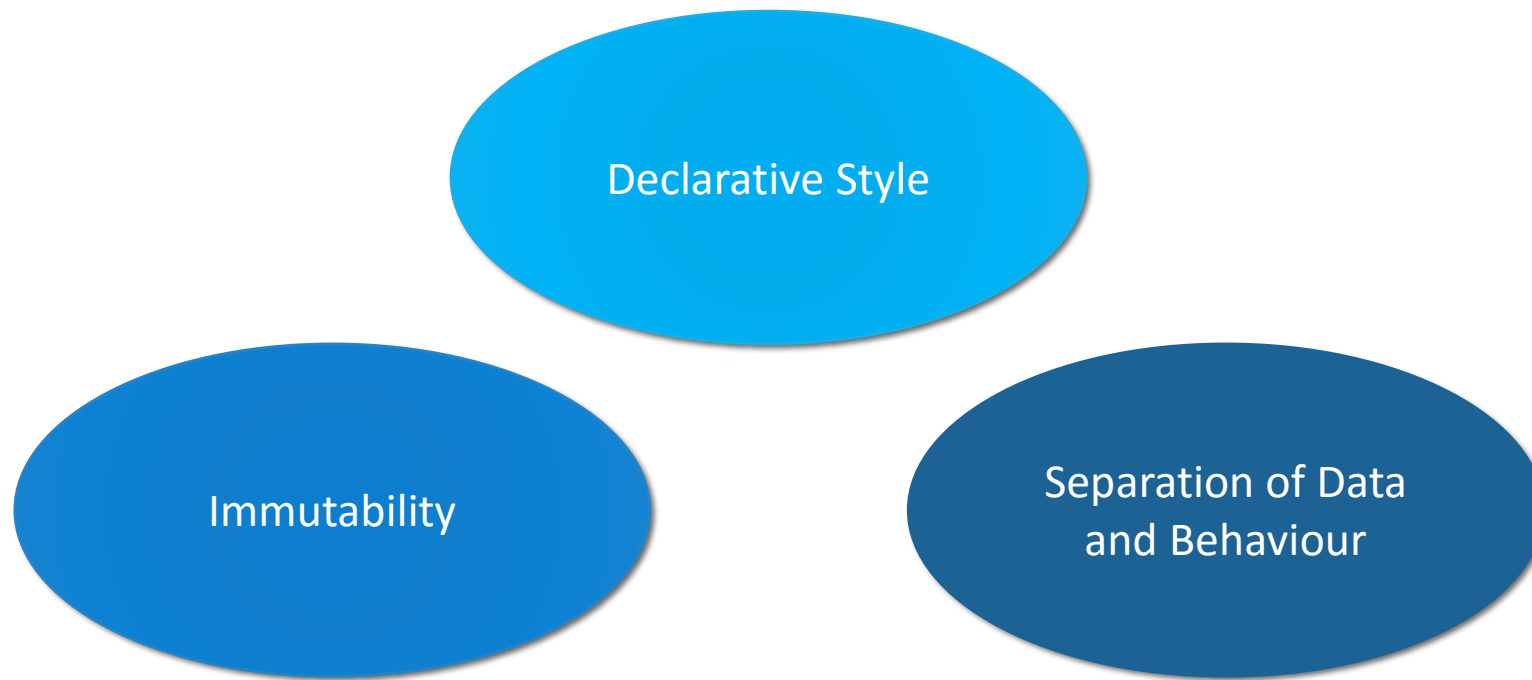
# Imperative vs Functional

C#

F#

Imperative                                        Functional

# Functional Core Concepts

# Bindings

let x = 1

let mutable x = 1
x <- 2

~~x = x + 1~~

let y = x + 1

# Functions

```
int Add(int x, int y)
{
    return x + y;
}
```

Func<int,int,int>

In    Out

```
let add x y = x + y
```

int -> int -> int

In    Out

let instead of
let insteadof
no parenthesis
noparenthesis
no return
no return
no braces
no braces

# Pure Functions and Side Effect

Impure Functions

Pure Function

let sum a b = a + b

```
let sum a b =
    let result = a + b
    saveResult result    ← Side Effect
    result
```

```
let mutable acc = 0
let sum a b =
    acc = acc + 1    ← Side Effect
    a + b
```

# Tuples

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

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

# Records

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

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

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

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

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

Structural Equality
Reference Types ←

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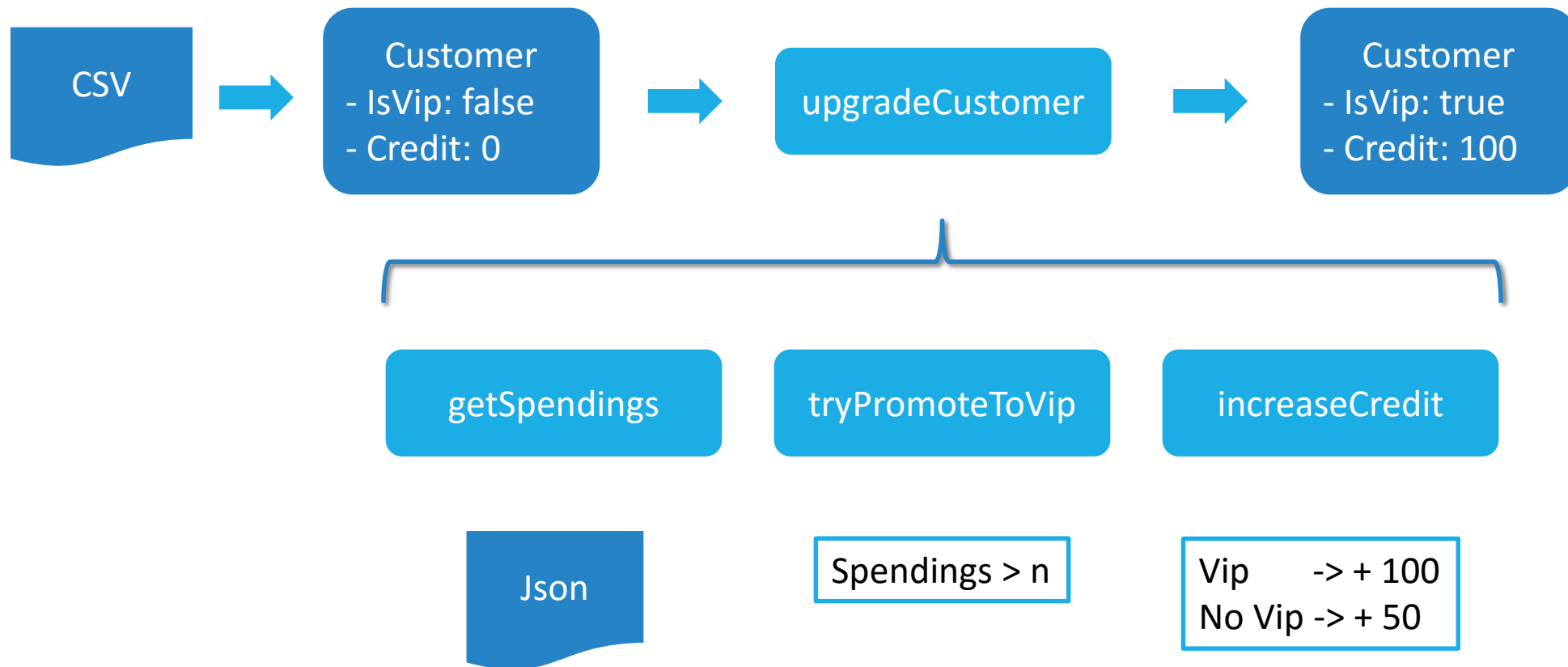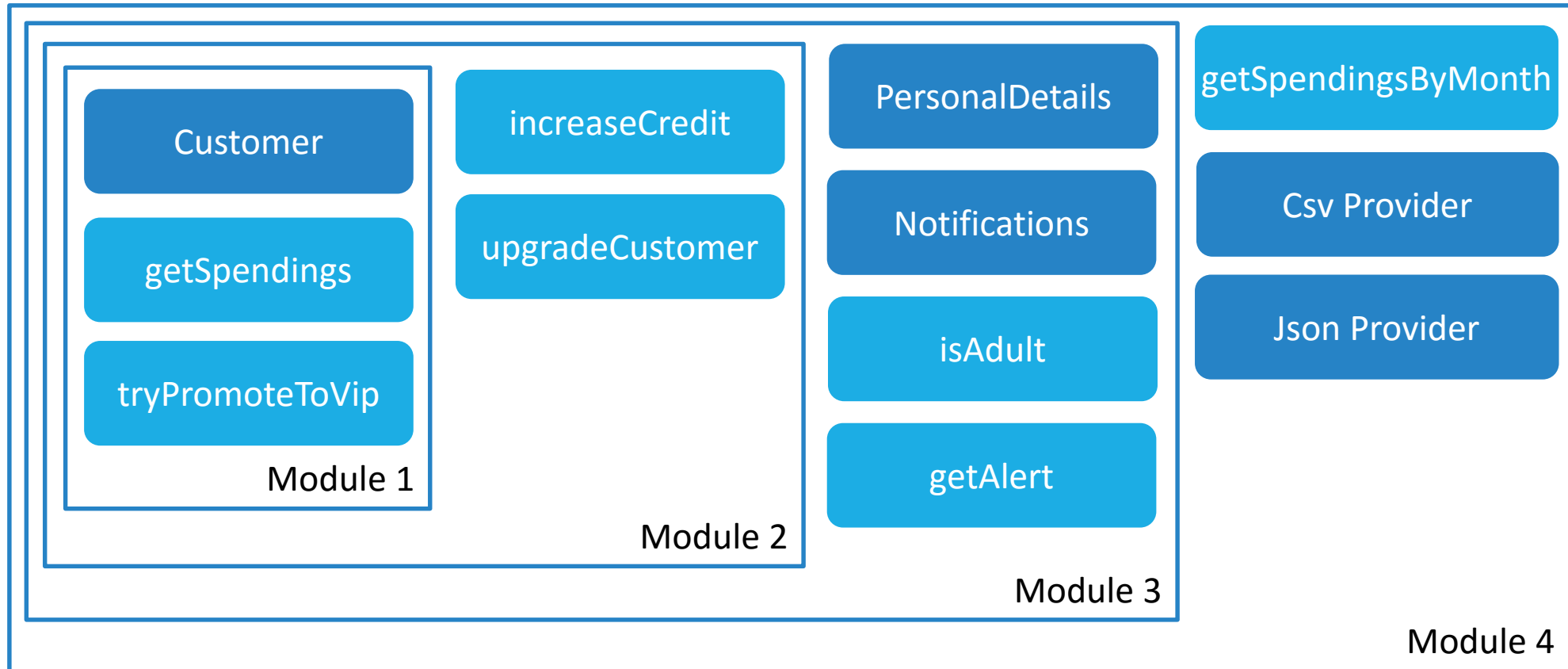
# Demo 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

# Exercise

CSV → Customer
- IsVip: false
- Credit: 0

→ upgradeCustomer → Customer
- IsVip: true
- Credit: 100

getSpendings     tryPromoteToVip     increaseCredit

Json     Spendings > n     Vip      -> + 100
                           No Vip -> + 50

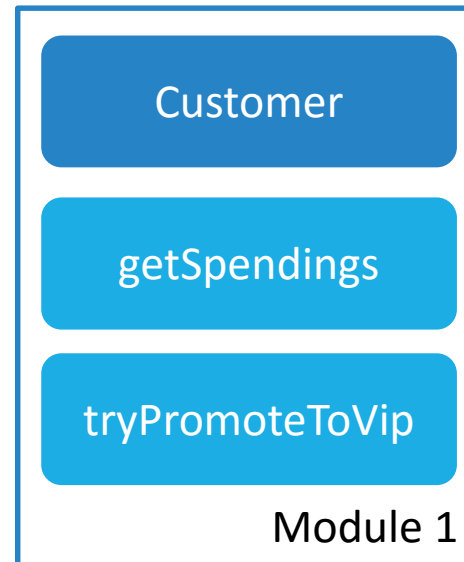# Exercise

# Exercise 1

# Exercise 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

# Review

> How do you return a value in a function?

> Can you explain this type?  string -> int -> object

> How do you change a Record?

# Module 2

HIGH ORDER FUNCTIONS | PIPELINING | PARTIAL APPLICATION | COMPOSITION

# High Order Functions

```
let sum (a: int) (b: int) = a + b
```

High Order Function

```
let compute (a: int) (b: int) (operation: int -> int -> int) =
    operation a b
```

High Order Function

```
let getOperation (type: OperationType) =
    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 (*)
```

# Pipelining Operator

```
let filter (condition: int -> bool) (items: int list) = // ...
```

```
let filteredNumbers = filter (fun n -> n > 10) numbers
```

```
let filteredNumbers = numbers |> filter (fun n -> n > 10)
```

```
let filteredNumbers = numbers
                        |> filter (fun n -> n > 10)
                        |> filter (fun n -> n < 20)
```

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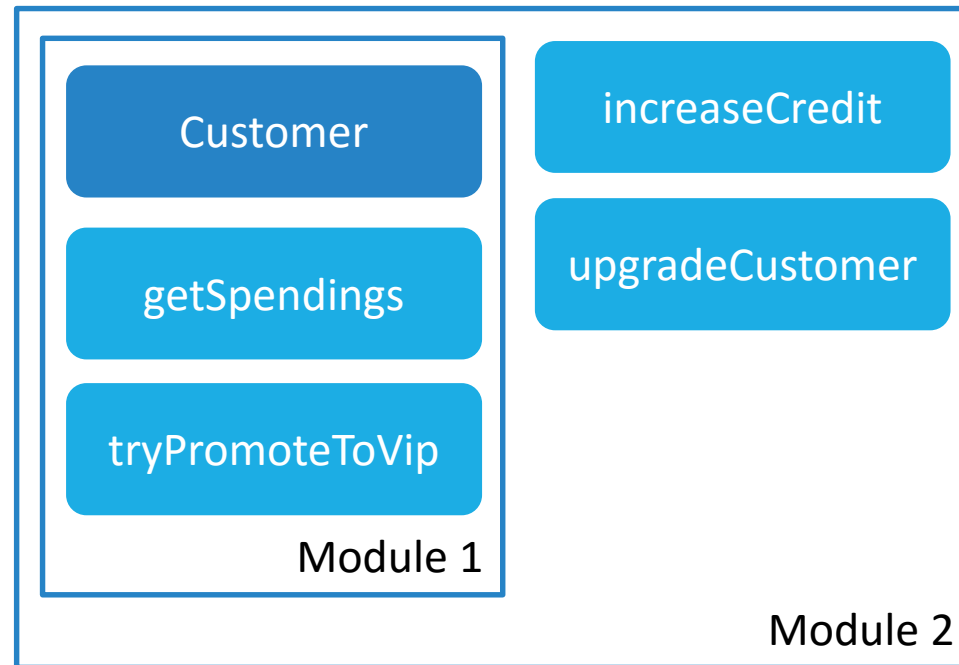
```
let addThree = addOne >> addTwo
```

```
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 is the benefit of using the pipelining operator?

> What happens when a function is called without its last parameter?

# Module 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

# NullReferenceExceptions (C#)

```
var customer = GetCustomerById(42);
```

```
var age = customer.Age;
```
NullReferenceException

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

Non Nullable
```
var age = GetCustomerAgeById(42);
```
```
public int GetCustomerAgeById(int id)
```

```
var result = GetCustomerAgeById(42);
var age = result.Value;
```
Hint: Possible Null

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

# Options

## C#

int

Int?

Customer

~~Customer?~~

## F#

int
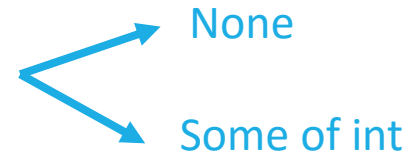
Int option → None
            → Some of int

Customer

Customer option → None
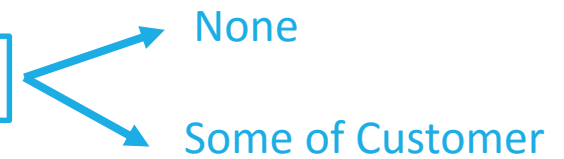                → Some of Customer

# Options

```
let divide x y = x / y
```
← int -> int

```
let divide x y =
    if y = 0 then None
    else Some(x / y)
```
← int -> int option

```
let result = divide 4 2
```
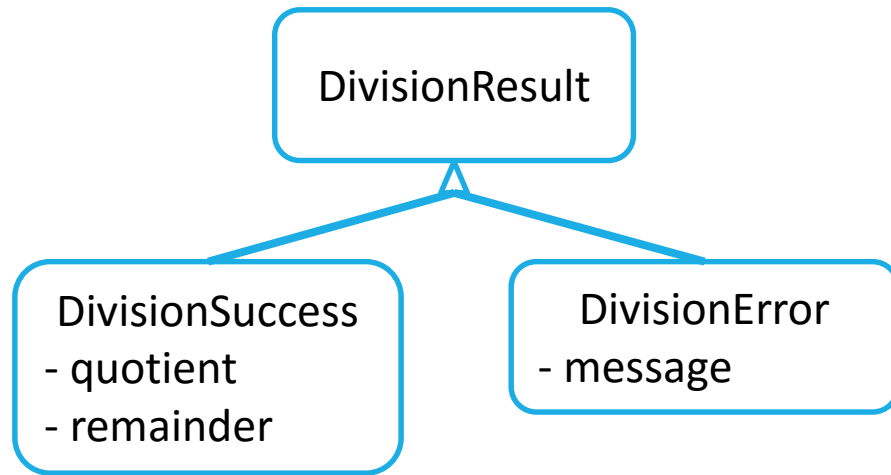← Some 2

```
let result = divide 4 0
```
← None

# Pattern Matching

```
let result = divide 4 0
if result = None then
    printfn "None"
else
    printfn "Result: %i" result.Value
```

```
let result = divide 4 0
match x with
| None -> printfn "None"
| Some n -> printfn "Result: %i" n
```

# Discriminated Unions



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

# Discriminated Unions

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

```
let result = divide 4 0
match result with
| DivisionSuccess (quotient, remainder) ->
    printfn "Quotient:%i Remainder:%i" quotient remainder
| DivisionError message ->
    printfn "Error: %s" message
```

# 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
```
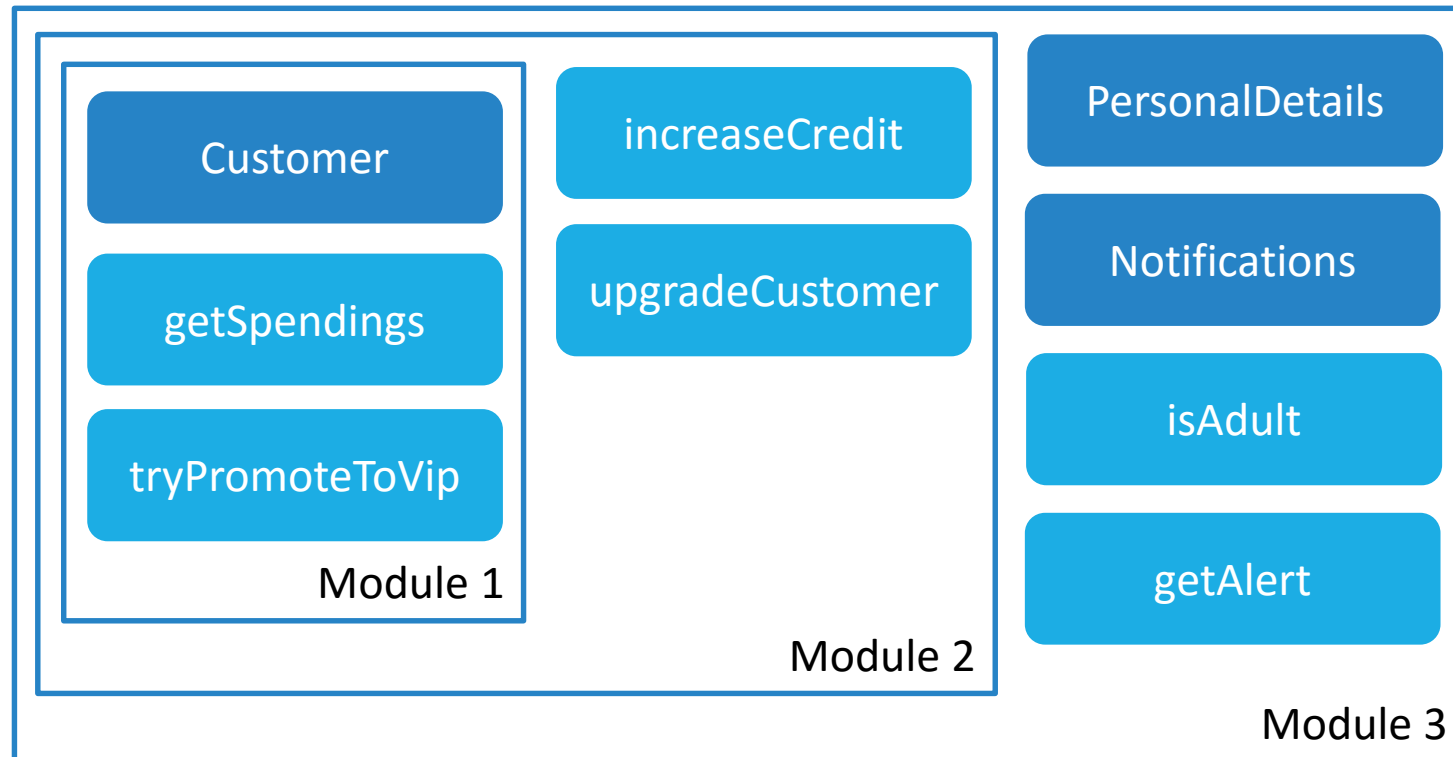← 6<m^2>

# Demo 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

# Exercise

# Exercise 3

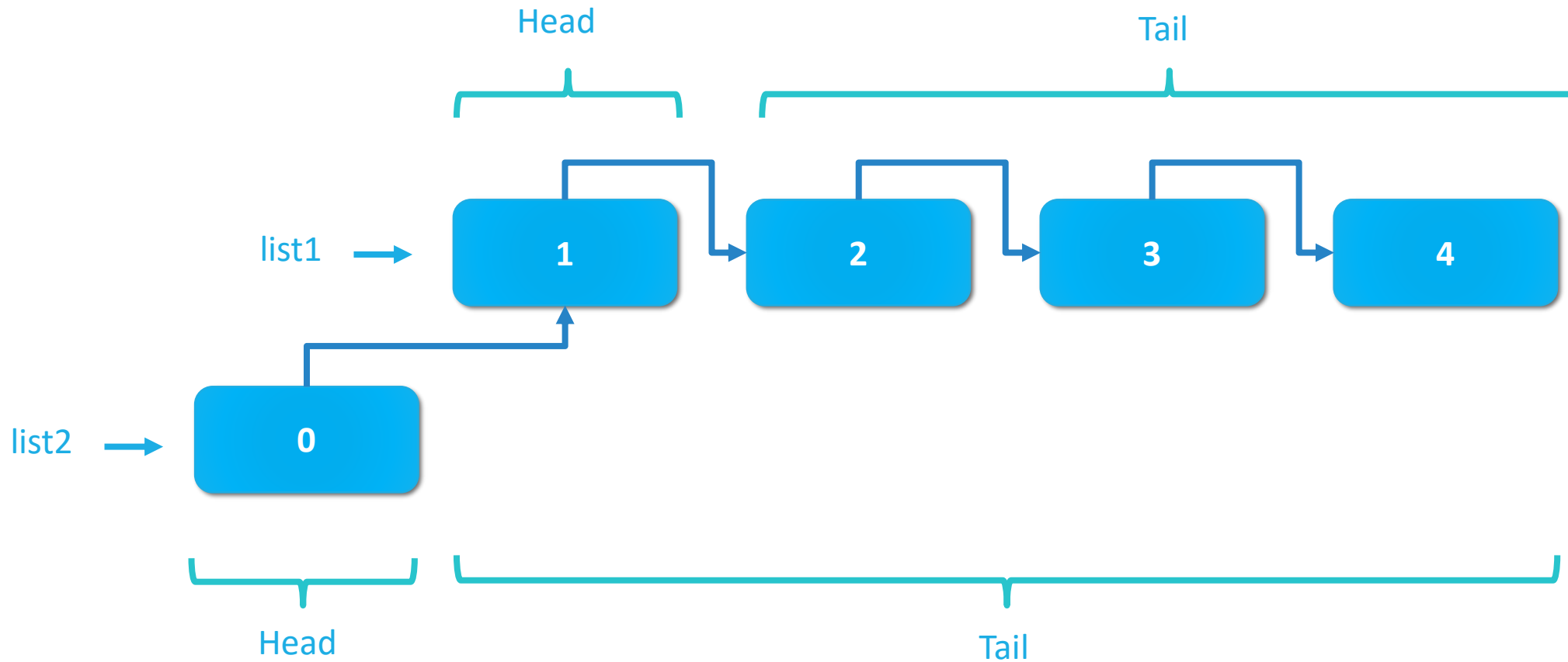OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS | UNITS OF MEASURE

# Review

> What happens if you multiply the same unit of measure?

>When should we use "_"?

> What are the possible types of string option?

# Module 4

FUNCTIONAL LISTS | OBJECT-ORIENTED PROGRAMMING | TYPE PROVIDERS

# Functional Lists

# Functional Lists

```
let numbers = [2; 3; 4]
```

```
let newNumbers = 1 :: numbers
```

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

```
let empty = []
```

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

```
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 }
```

# List Module

## F#

## C#

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

| F# | C# |
|---|---|
| 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

# Object Oriented Programming

## Immutable Fields

```
type MyClass(myField: int) =

    member this.MyProperty = myField


    member this.MyMethod methodParam =
        myField + methodParam
```

## Mutable Fields

```
type MyClass(myField: int) =
    let mutable myMutableField = myField

    member this.MyProperty
        with get () = myMutableField
        and set(value) = myMutableField <- value


    member this.MyMethod methodParam =
        myField + methodParam
```

# Object Expressions

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

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

# Type Providers
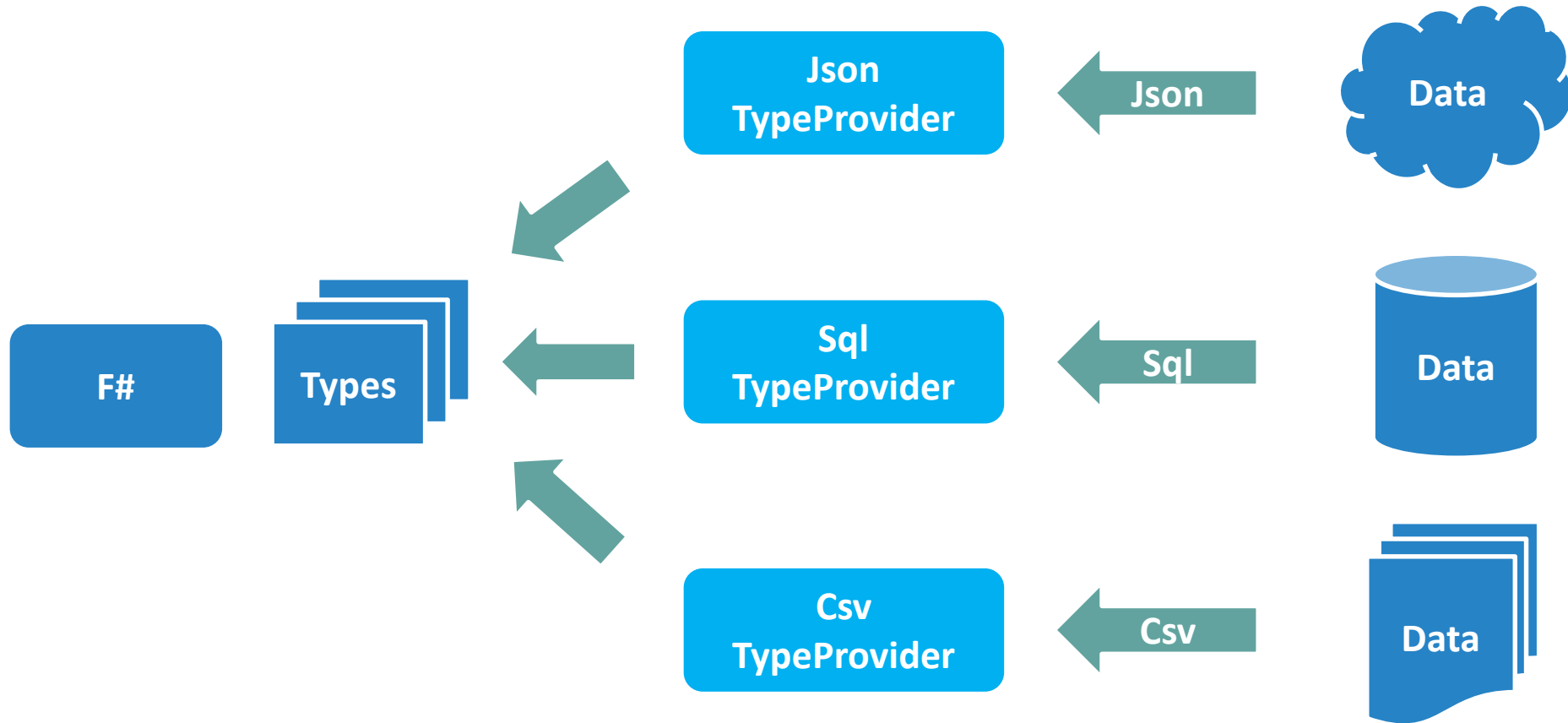
# CSV Type Provider

```
type Customer = CsvProvider<"sample.csv">
let customers = Customer.Load "real.csv"

customers.Rows
|> Seq.iter (fun r -> printfn "%s: $%g" r.Name r.Credit)
```

sample.csv

```
Id,Name,IsVip,Credit
1,Customer1,false,0.0
```

real.csv

```
Id,Name,IsVip,Credit
1,Customer1,false,0.0
2,Customer2,false,10.0
3,Customer3,false,30.0
4,Customer4,true,50.0
```

# Type Providers

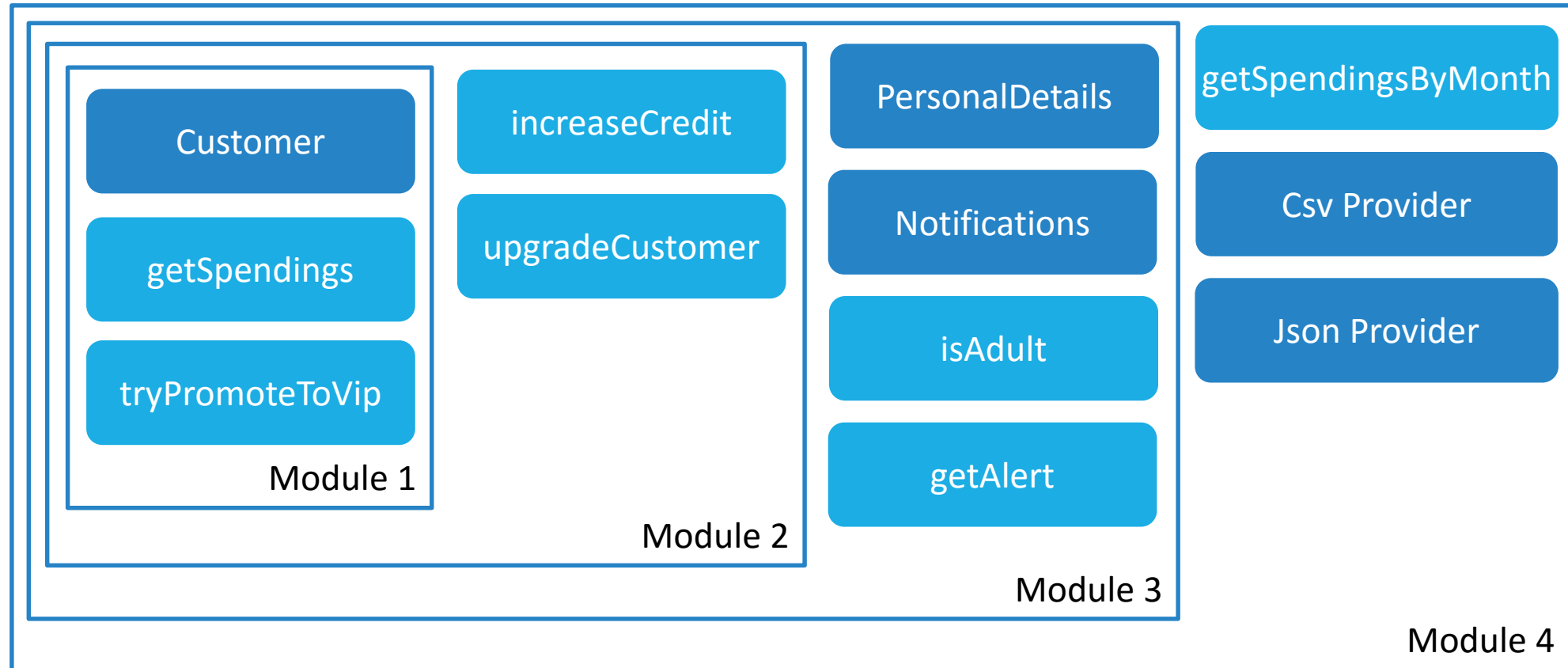| | | | | |
|---|---|---|---|---|
| CSV | Json | XML | SQL | EF |
| Azure Storage | OData | Excel | R | Reflection |
| WMI | LINQ2SQL | Hadoop / Hive | Freebase | WSDL |

**And many more**

# Demo 4

FUNCTIONAL LISTS | OBJECT-ORIENTED PROGRAMMING | TYPE PROVIDERS

# Exercise 4

FUNCTIONAL LISTS | OBJECT-ORIENTED PROGRAMMING | TYPE PROVIDERS

# Review

> Which keyword do we use to declare a class property or method?

> Why do we refer to "Data.json" twice?

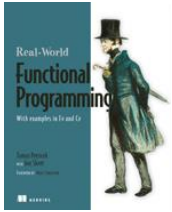> What happens if I change the name of a column in the sample.json file?

# Thank you

JORGE FIORANELLI - @JORGEFIORANELLI

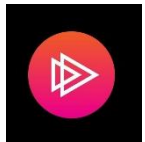# Resources

fsharp.org / c4fsharp.net

Real-World Functional Programming
By Tomas Petricek

Scott Wlaschin       fsharpforfunandprofit.com
                     fpbridge.co.uk/why-fsharp.html

FSHARP TV  fsharp.tv

pluralsight.com/search?q=f%23&categories=all

Skills Matter: skillsmatter.com (tag: f#)