Functional F# Programming in .NET a success story

Riccardo Terrell

V3.2

Objectives

> Understand the basic core principles behind FP
> Understand the F# syntax and structures
> Get motivation to practice and master F#
> How to build a DSL in F#
> Functional parallel programming (bonus)

Pre-requisites

> Windows

- dotnet core
- Visual Studio 2017/2019
- Rider (JetBrains)
- Visual Studio Code
  - C# Extensions
  - F# Compiler + Ionide package (optional)

>Linux

- Visual Studio Code + dotnet core + Ionide package

>Mac

- Visual Studio for Mac + or dotnetcore
- Visual Studio Code + (Mono or dotnetcore) + Ionide package
- Rider (JetBrains)

Download links:
https://github.com/rikace/codemash-fsharpws

See README section pre-requisites

Download links:

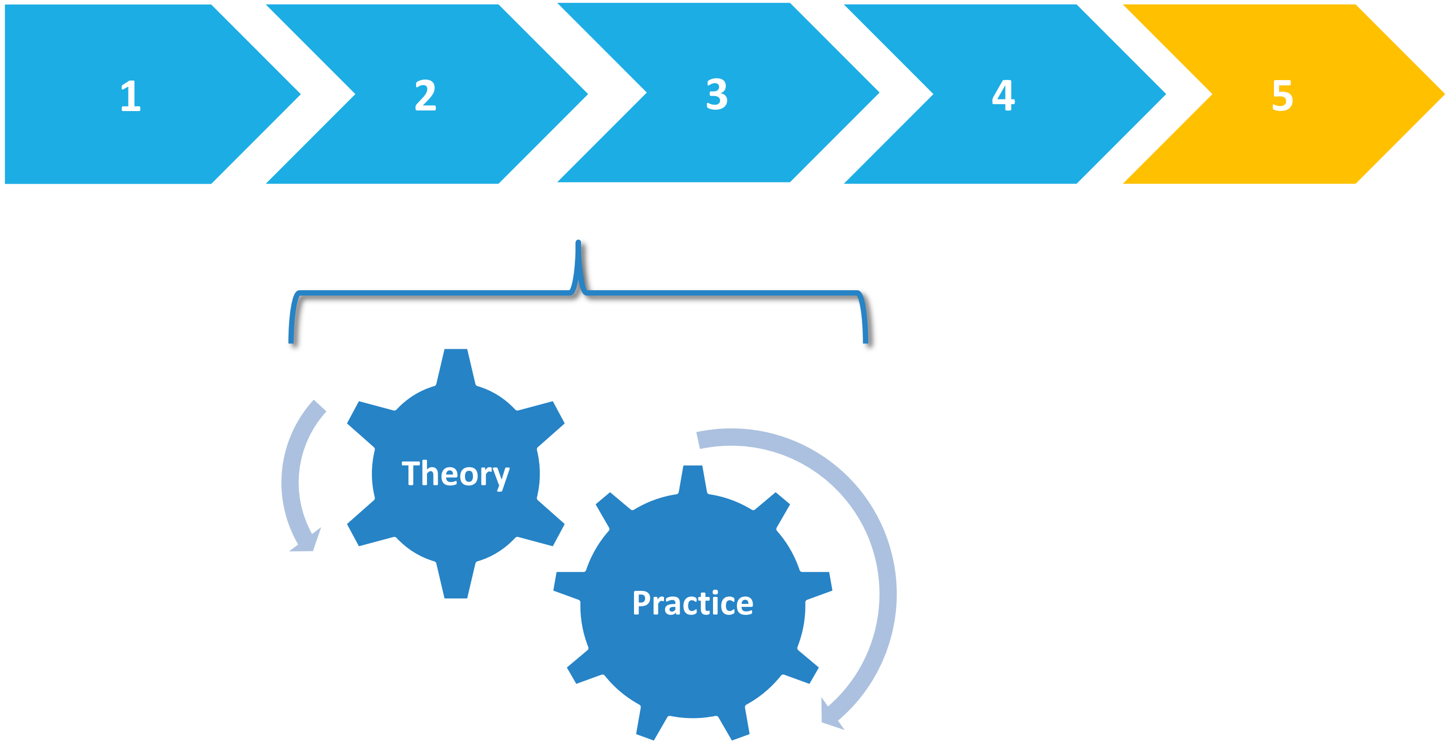
https://github.com/rikace/codemash-fsharpws

See README section pre-requisites

Disclaimer

- Let’s keep the session interactive
- Skipping slides
- After this class you will need to keep practicing
- This is not just an introduction
- The code is not production-ready

Modules



Agenda

Intro

What is F# and the tenets of functional programming

Module 1

Bindings | Functions | Tuples | Records

Module 2

High order functions | Pipelining | Partial application | Composition

Module 3

Options | Pattern matching | Discriminated unions

Module 4

Functional lists | DSL

Module 5

Concurrency | Async Programming | Agents

Module 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

Bindings

```
let x = 1
```

~~x = x + 1~~

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

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

Functions

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

```
Func<int,int,int>
```

In Out

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

```
int -> int -> int
```

In Out

let instead of

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 }
```

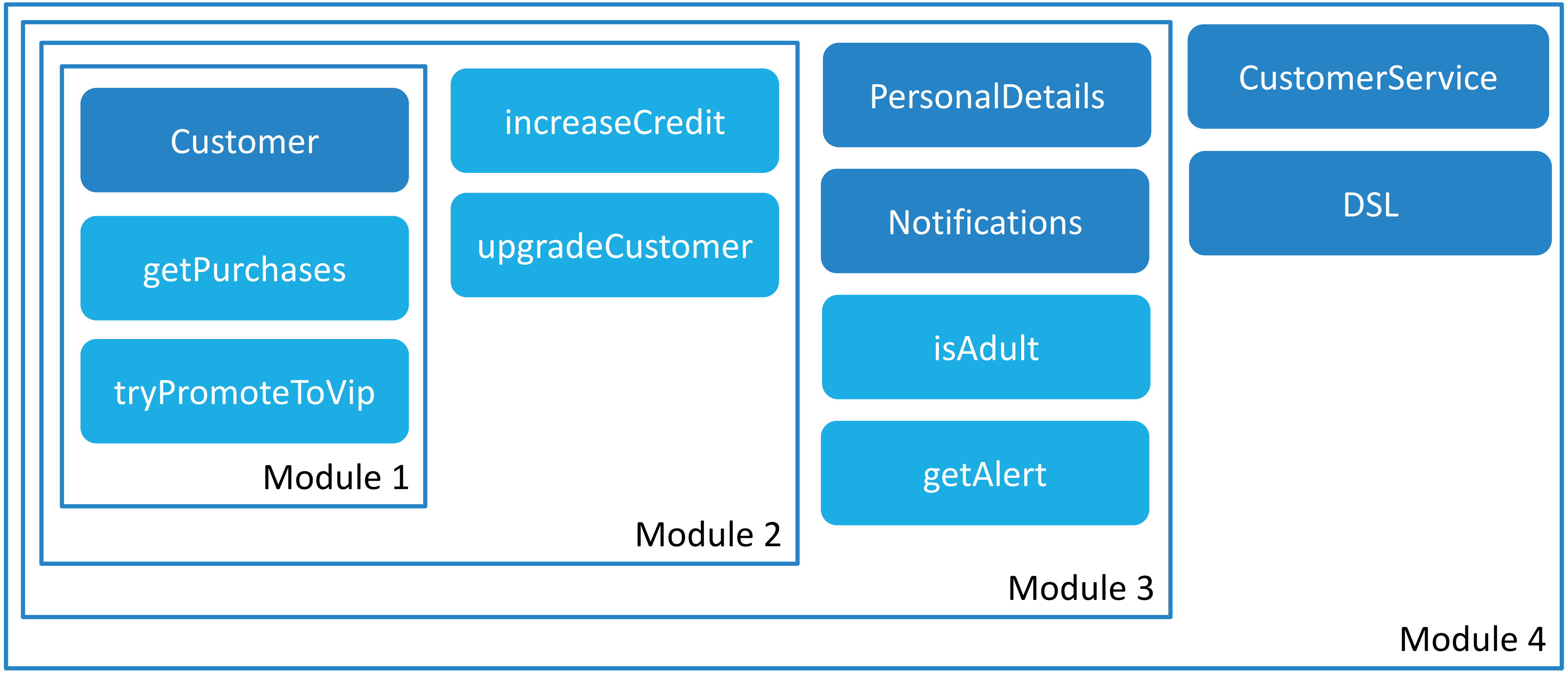
Structural Equality
Reference Types

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

Demo 1

BINDINGS | FUNCTIONS | TUPLES | RECORDS

Exercise


Customer
getPurchases
tryPromoteToVip
Module 1
increaseCredit
upgradeCustomer
Module 2
PersonalDetails
Notifications
isAdult
getAlert
Module 3
CustomerService
DSL
Module 4

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)
```

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

Partial Application

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

```
let result = sum 1 2
```

Returns int = 3

```
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
```

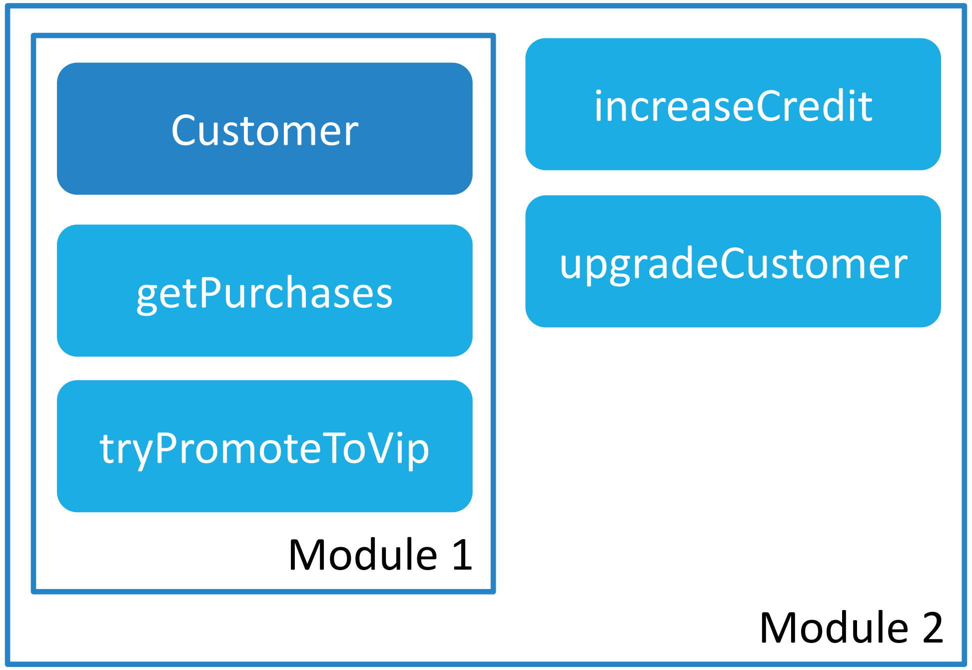
```
let result = addThree 1
```

Returns int = 4

Demo 2

HIGH ORDER FUNCTIONS | PIPELINING | PARTIAL APPLICATION | COMPOSITION

Exercise 2


Customer
getPurchases
tryPromoteToVip
Module 1
increaseCredit
upgradeCustomer
Module 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

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 -> int

```
let divide x y =
    if y = 0 then None
    else Some(x / y)
```

← int -> int -> int option

```
let result = divide 4 2
```

← Some 2

```
let result = divide 4 0
```

← None

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

Nested null checks

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

Nulls are a code smell: replace with Maybe!

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse x.Value
        if y.IsSome then
            let z = doAThirdThing y.Value
            if z.IsSome then
                let result = z.Value
                result
            else
                null
        else
            null
    else
        null
```

Much more elegant, yes?

No! This is ugly!
But there is a pattern we can exploit...

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse x.Value
        if y.IsSome then
            let z = doAThirdThing y.Value
            if z.IsSome then
                // do something with z.Value
                // in this block
            else
                None
        else
            null
    else
        null
```

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse x.Value
        if y.IsSome then
           // do something with z.Value
           // in this block





        else
            None
    else
        null
```

Pyramid of doom : null testing

```
let example input =
    let x = doSomething input
    if x.IsSome then
      // do something with z.Value
      // in this block









    else
        None
```

Pyramid of doom : null testing

```
if opt.IsSome then
    //do something with opt.Value
else None
```

```
let ifSomeDo (f:a -> Option<b>) (opt: Option<a>) =
    if opt.IsSome then
        f( opt.Value )
    else
        None
```

```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo (fun z -> Some z)
```

```
doSomething(input)
.ifSomeDo(doSomethingElse)
.ifSomeDo(doAThirdThing)
```

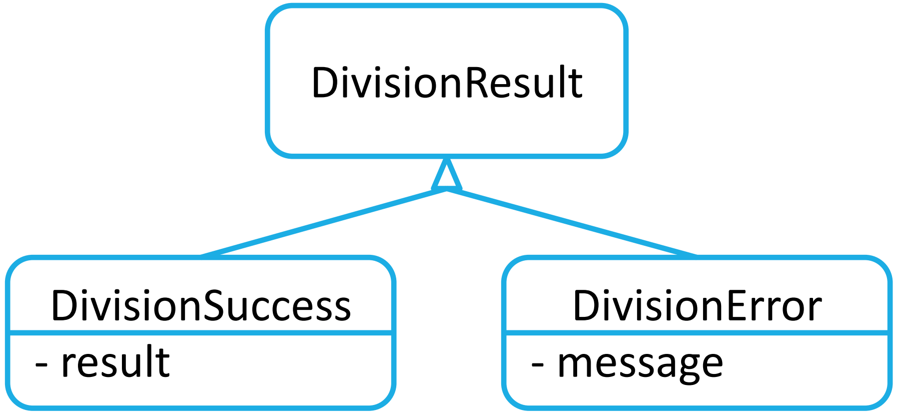
Pattern Matching

```
let result = divide 4 0
if result = None then
   printfn "No Result"
else
   printfn "Result: %i" result.Value
```

```
let result = divide 4 0
match result with
| None -> printfn "No Result"
| Some n -> printfn "Result: %i" n
```

Discriminated Unions

```
type Boolean =
    | True
    | False
```




```
type DivisionResult =
    | DivisionSuccess of result : int
    | DivisionError of message : string
```

Discriminated Unions

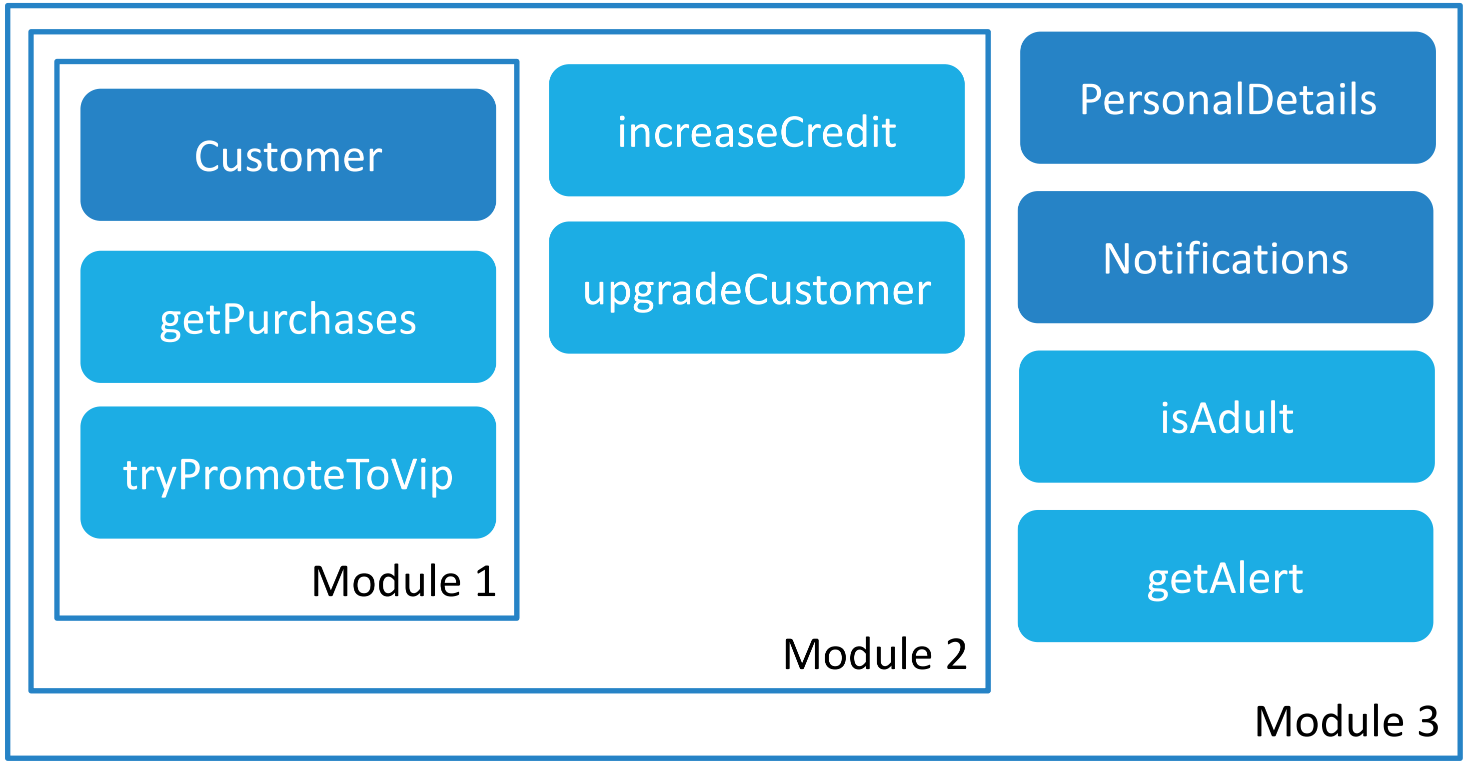
```
let divide x y =
    match y with
    |0 -> DivisionError("Divide by zero")
    |_ -> DivisionSuccess(x / y)
```

```
let result = divide 4 0
match result with
| DivisionSuccess result -> printfn “Result: %i" result
| DivisionError message -> printfn "Error: %s" message
```

Demo 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS

Exercise


Customer
getPurchases
tryPromoteToVip
Module 1
increaseCredit
upgradeCustomer
Module 2
PersonalDetails
Notifications
isAdult
getAlert
Module 3

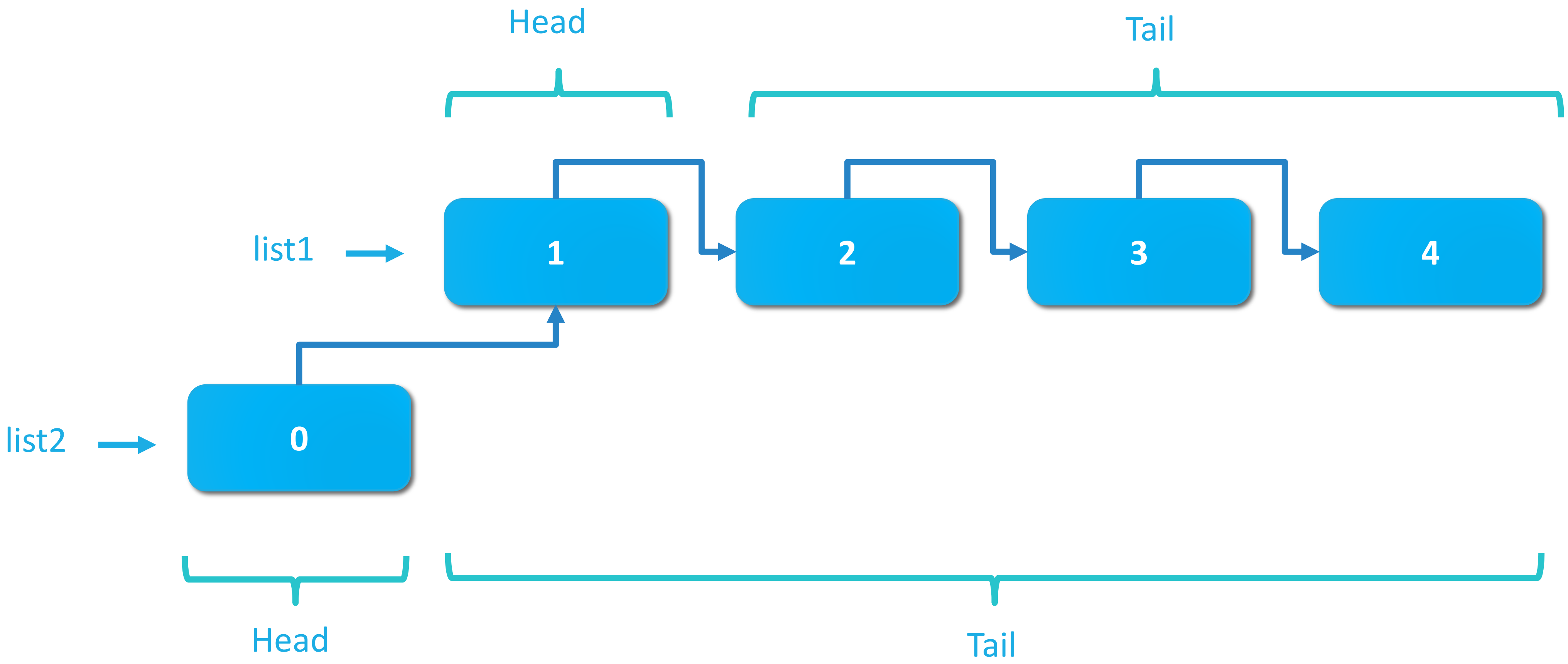
Exercise 3

OPTIONS | PATTERN MATCHING | DISCRIMINATED UNIONS

Module 4

FUNCTIONAL LISTS | OBJECT EXPRESSION | DSL

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

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

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

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

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

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

DSL = model + syntax

How a DSL is defined

- Primitives (data elements)
- Combinators
- Semantic & Syntax

Why use DSL

- Domain Focus
  - Non-experts can read it
- Productivity
- Reliability
- Correctness
- Maintainability
  - Easier to reason

Hides the implementation

Domain-specific language approach

We have a class of problems

Create a language for the class
Use language to solve them

Domain model

Understand the problem domain!
Using ADT - discriminated unions

Domain-specific language

Primitives – basic building blocks
Composition – how to put them together

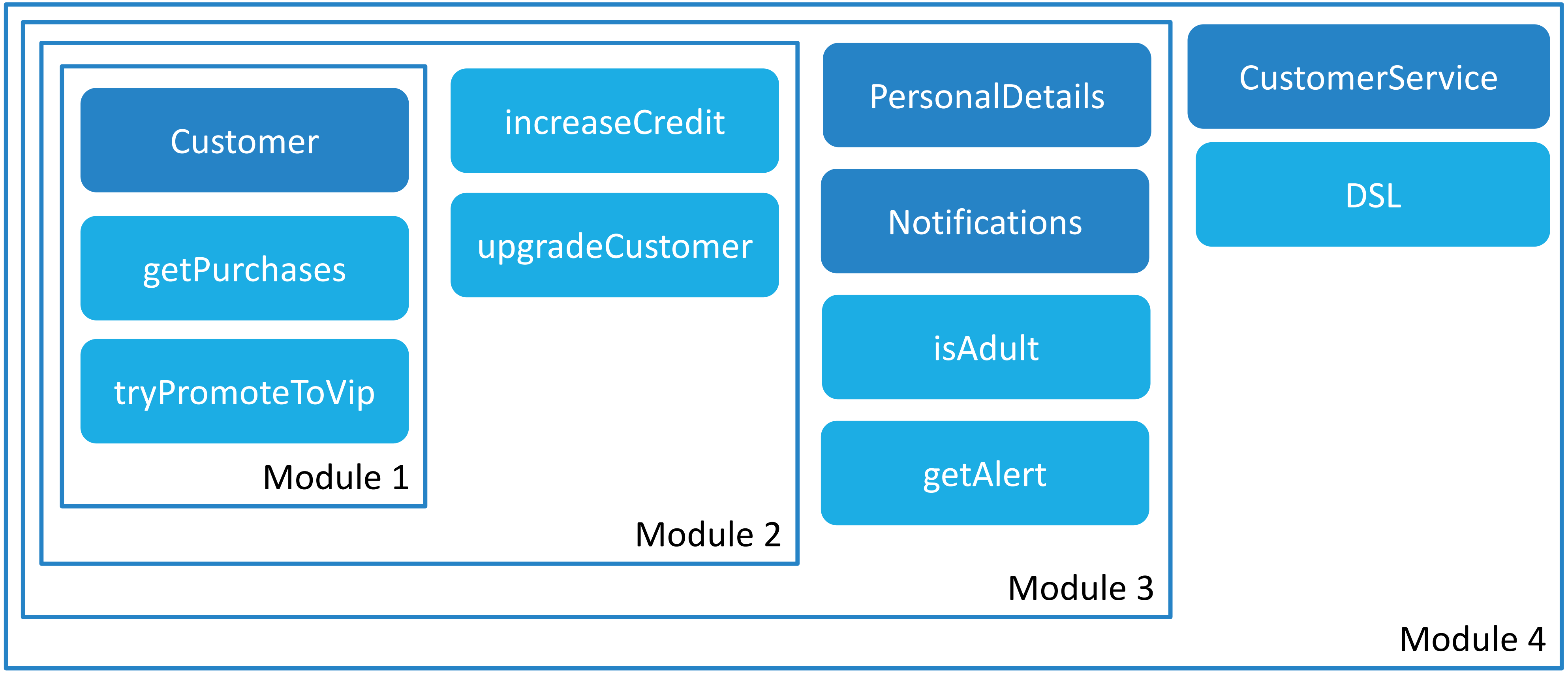
Demo 4

FUNCTIONAL LISTS | OBJECT EXPRESSION | DSL

DSL
ordering a cup of coffee

```
type size = Tall | Grande | Venti
type drink = Latte | Cappuccino | Mocha
type extra = Shot | Syrup
type Cup = { Size:size; Drink:drink; Extras:extra list }
    static member (+) (cup:Cup,extra:extra) =
                  { cup with Extras = extra :: cup.Extras }
    static member Of size drink =
                  { Size=size; Drink=drink; Extras=[] }

let price (cup:Cup) =
    let tall, grande, venti =
        match cup.Drink with
        | Latte -> 2.69, 3.19, 3.49
        | Cappuccino -> 2.69, 3.19, 3.49
        | Mocha -> 2.99, 3.49, 3.79
    let basePrice =
        match cup.Size with
        | Tall -> tall
        | Grande -> grande
        | Venti -> venti
    let extras =
       cup.Extras |> List.sumBy (function
                                 | Shot -> 0.59
                                 | Syrup -> 0.39 )
   basePrice + extras
```

Exercise 4


Customer
getPurchases
tryPromoteToVip
Module 1
increaseCredit
upgradeCustomer
Module 2
PersonalDetails
Notifications
isAdult
getAlert
Module 3
CustomerService
DSL
Module 4

Exercise 4

FUNCTIONAL LISTS | OBJECT EXPRESSION | DSL

Module 5

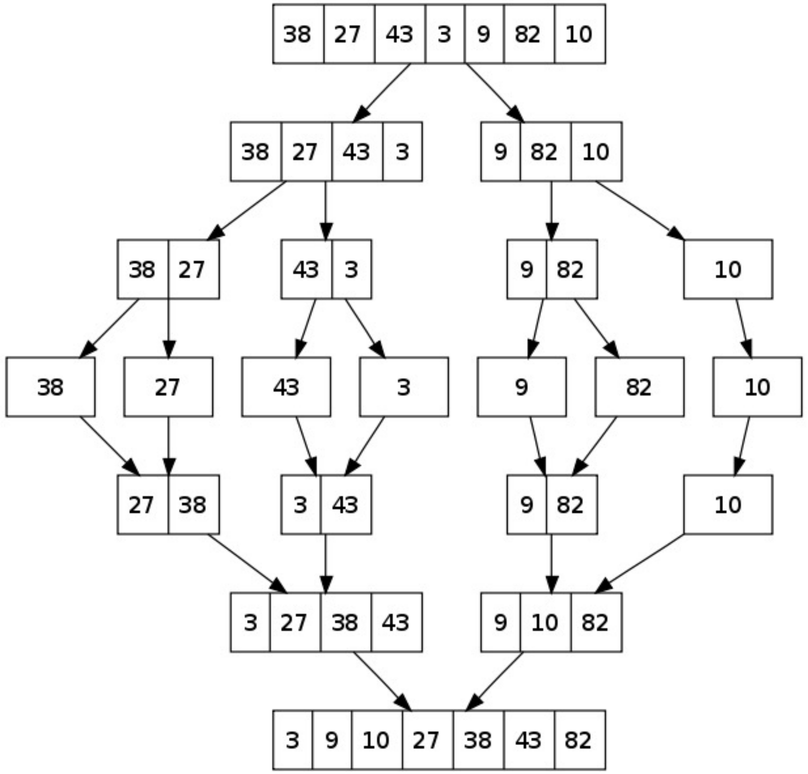
CONCURRENCY | ASYNC PROGRAMMING | MAILBOXPROCCESOR

Its about maximizing resource use

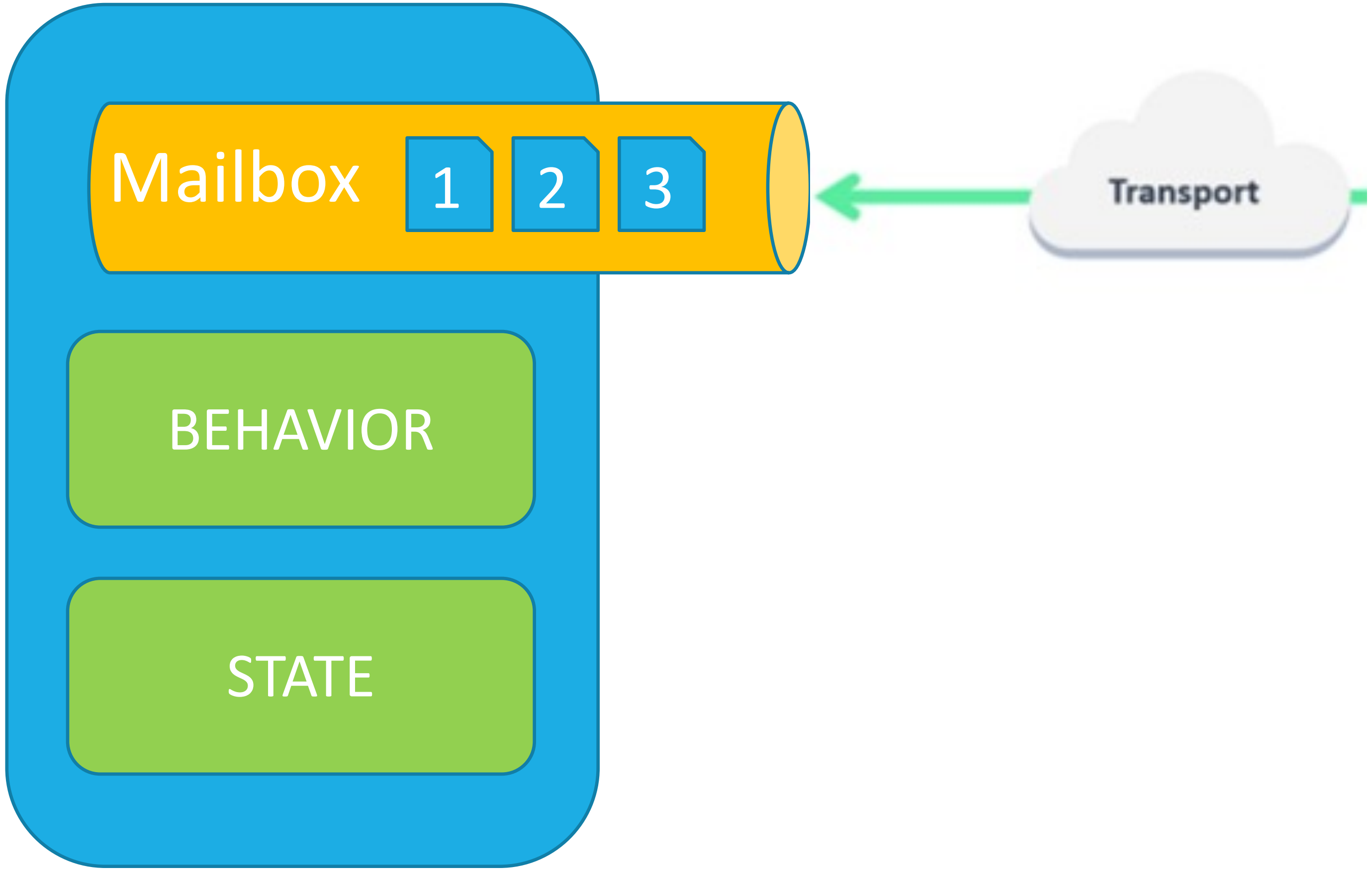
To get the **best** performance, your application has to partition and divide processing to take full advantage of multicore processors – enabling it to do **multiple** things at the same time, i.e. **concurrently**.

```
void QuickSort_Parallel<T>(T[] items, int left, int right)
{
    int pivot = left;


    SwapElements(items, left, pivot);

    Parallel.Invoke(
            () => QuickSort_Parallel(items, left, pivot - 1),
            () => QuickSort_Parallel(items, pivot + 1, right)
        );
}
```

Divide and Conquer



Message Passing based concurrency



- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object
- Running on it’s own thread
- No shared state
- Messages are kept in mailbox and processed in order
- Massively scalable and lightening fast because of the small call stack

Demo 5

CONCURRENCY | ASYNC PROGRAMMING | AGENT

Exercise 5

PARALLEL WEB CRAWLER


That's all Folks!