Functional F# Programming in .NET a success story

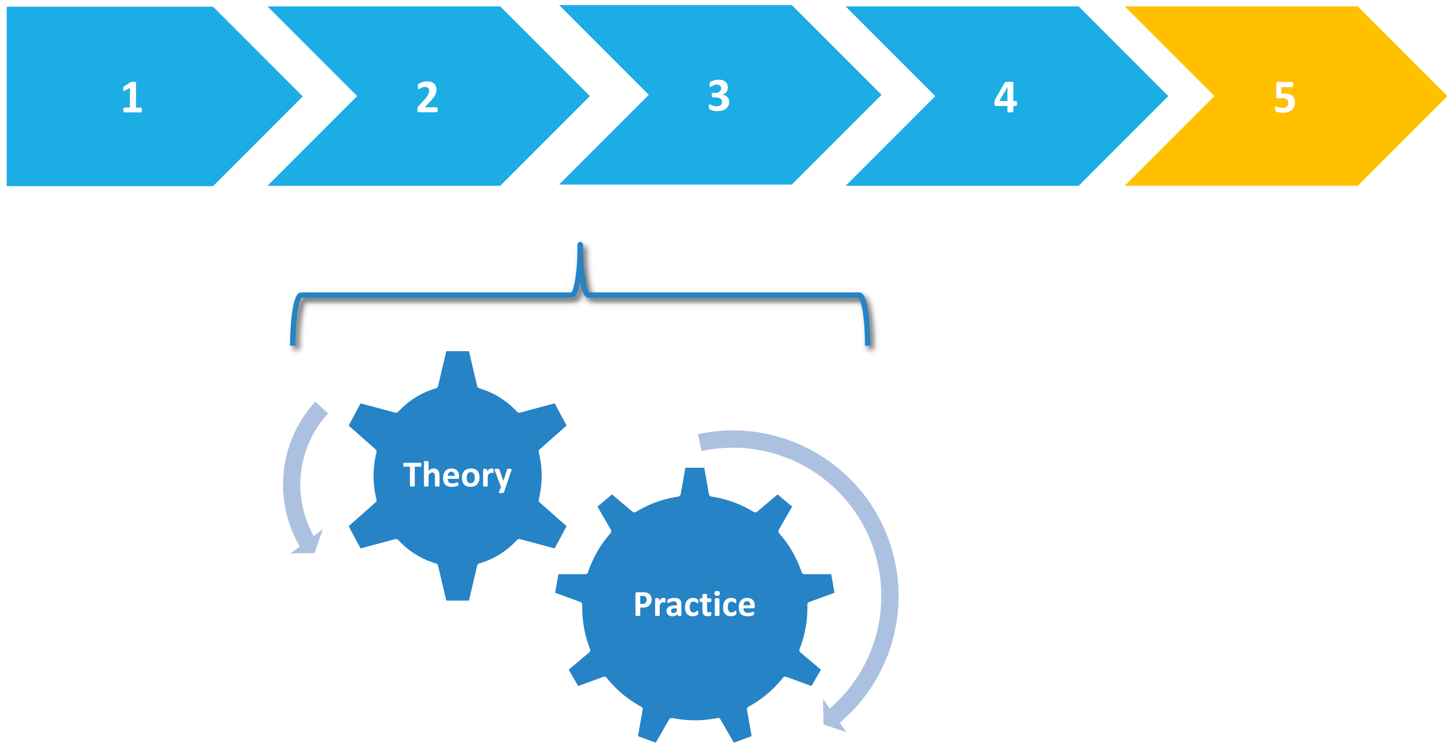
Riccardo Terrell

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)

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

```
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

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 }
```

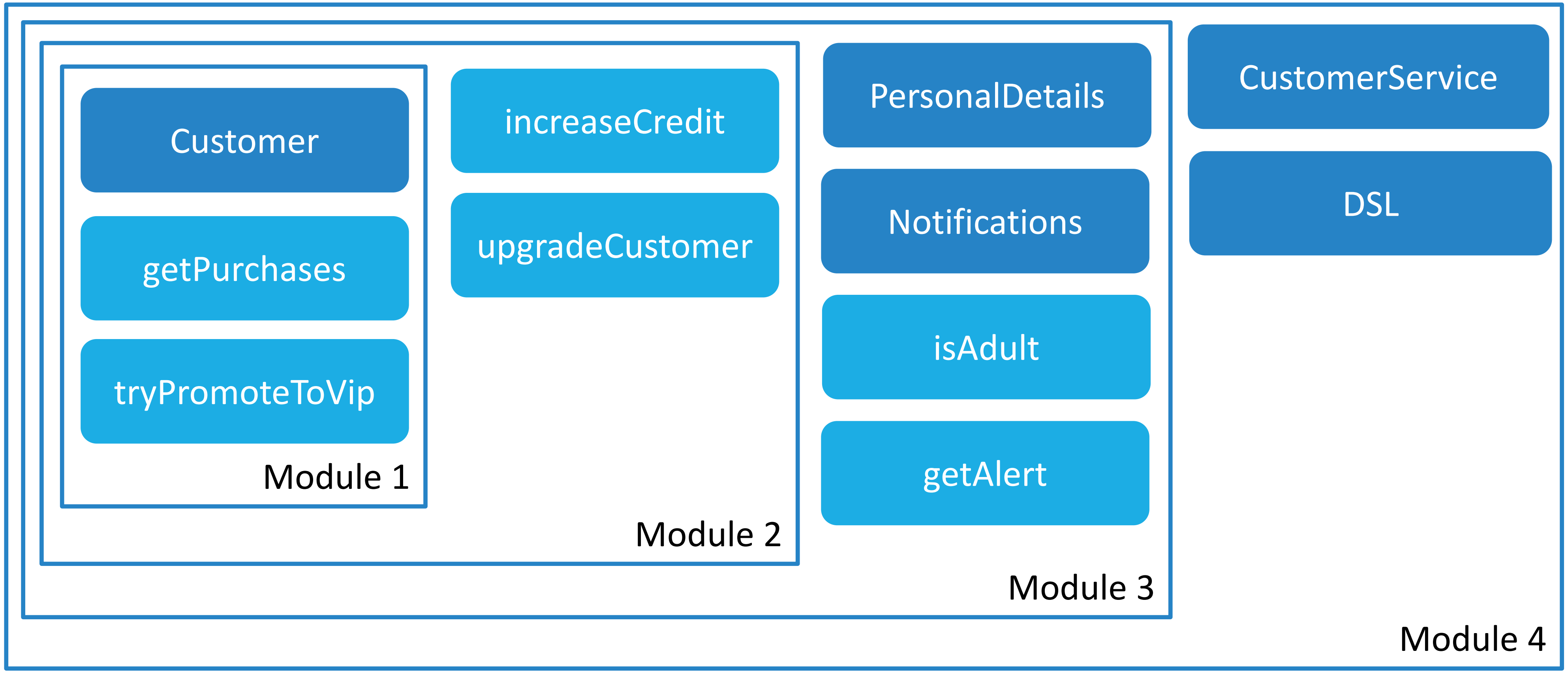
```
let result1 = { Quotient = 3; Remainder = 1 }
let result2 = { Quotient = 3; Remainder = 1 }
result1 = result2 // true
```

Structural Equality
Reference Types

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
```

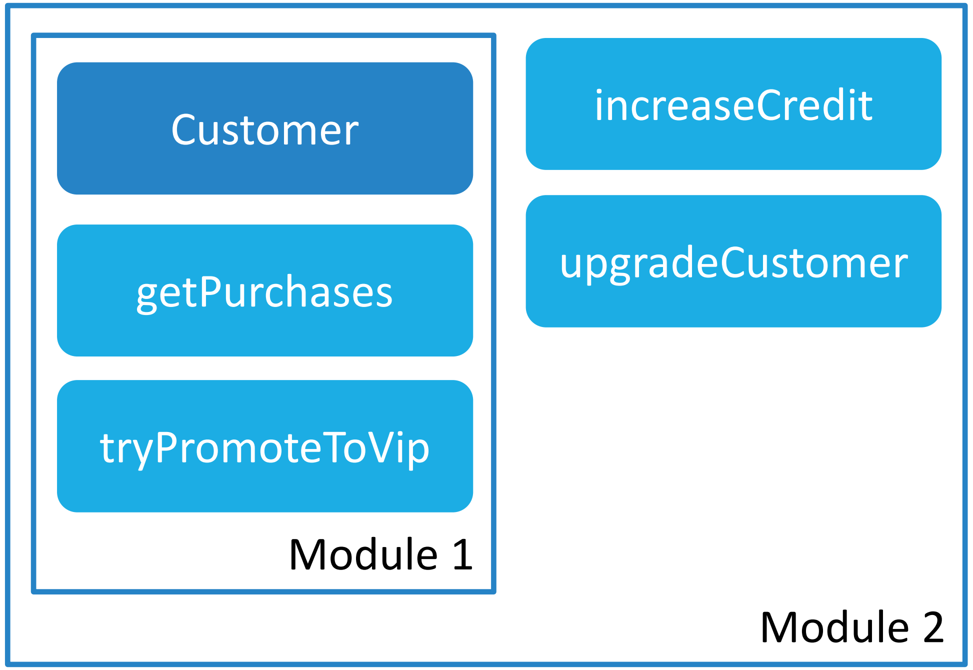
```
let result = addThree 1
```

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

```
var age = GetCustomerAgeById(42);
```

```
var result = GetCustomerAgeById(42);
var age = result.Value;
```

Hint: Possible Null

Non Nullable

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

```
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)
```

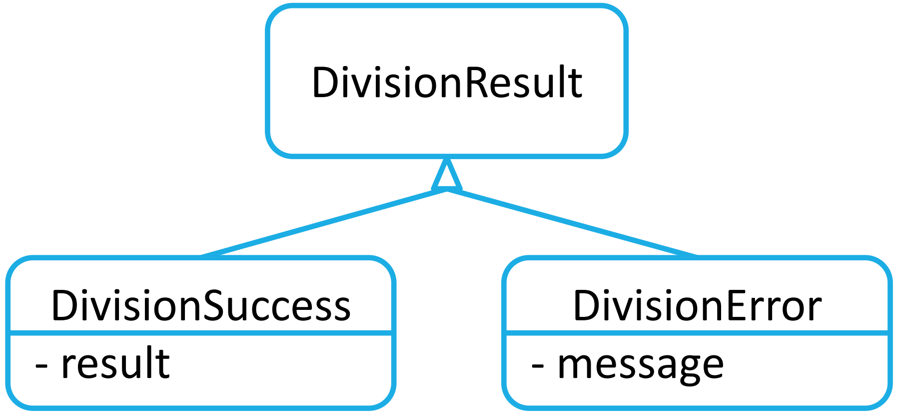
Pattern Matching

```
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

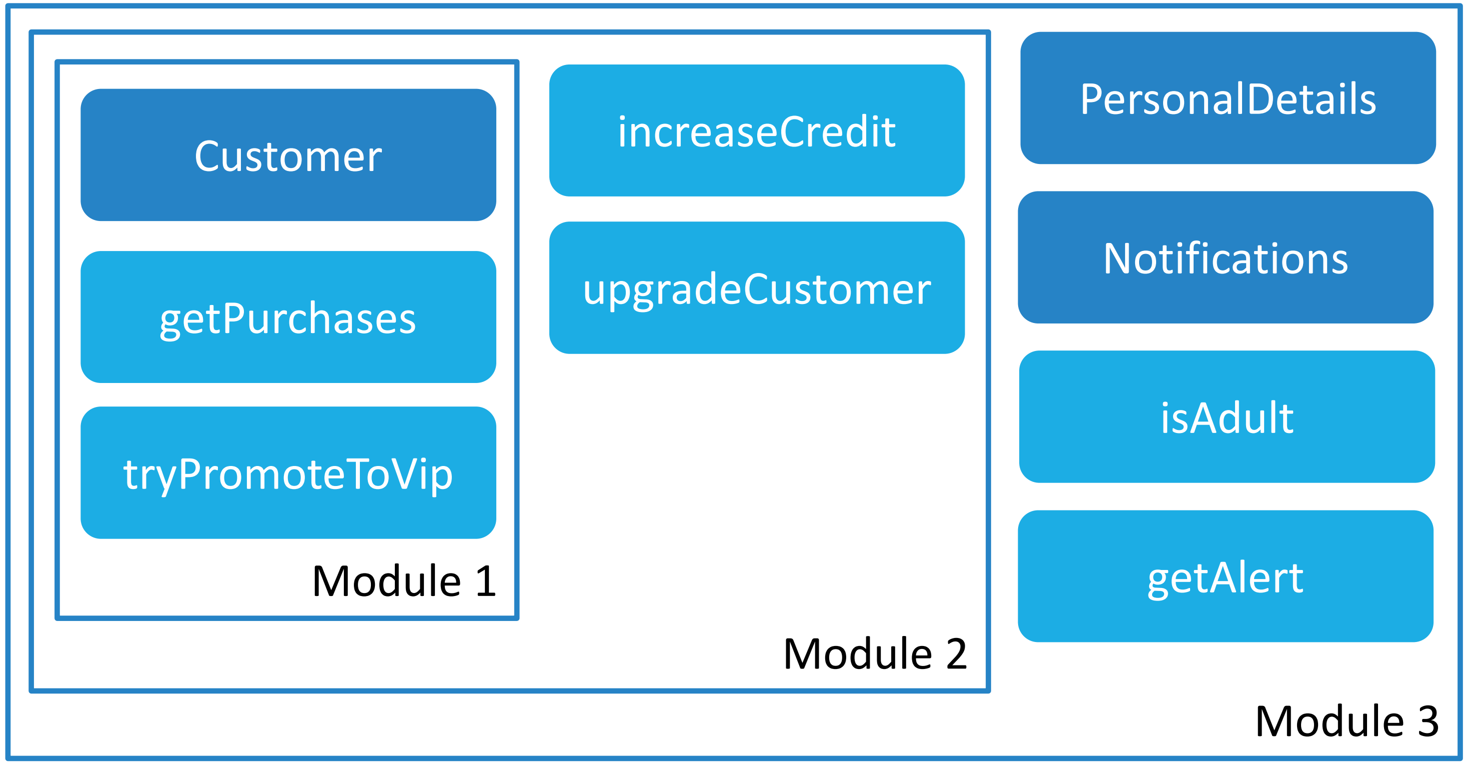
```
let divide x y =
   match y with
   |0 -> DivisionError("Divide by zero")
   |_ -> DivisionSuccess(x / y)
```

```
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

Review

- When should we use “_”?
- What are the possible types of string option?
- What happens when a function is called without its last parameter?

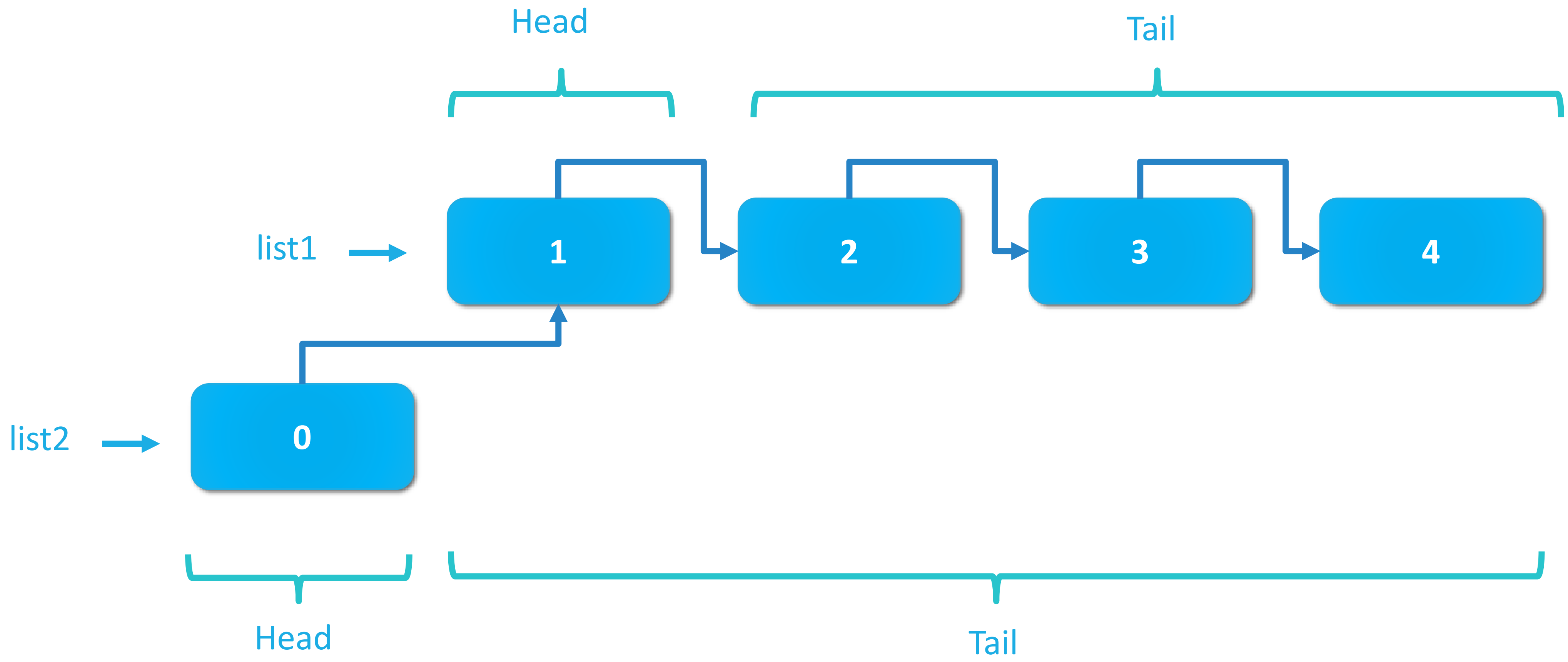
Pattern Matching Guards

```
let divide x y =
   match y with
   |0 -> DivisionError("Divide by zero")
   |_ when x > 1000 ->  DivisionSuccessForLargeNumber(x / y)
   |_  -> DivisionSuccess(x / y)
```

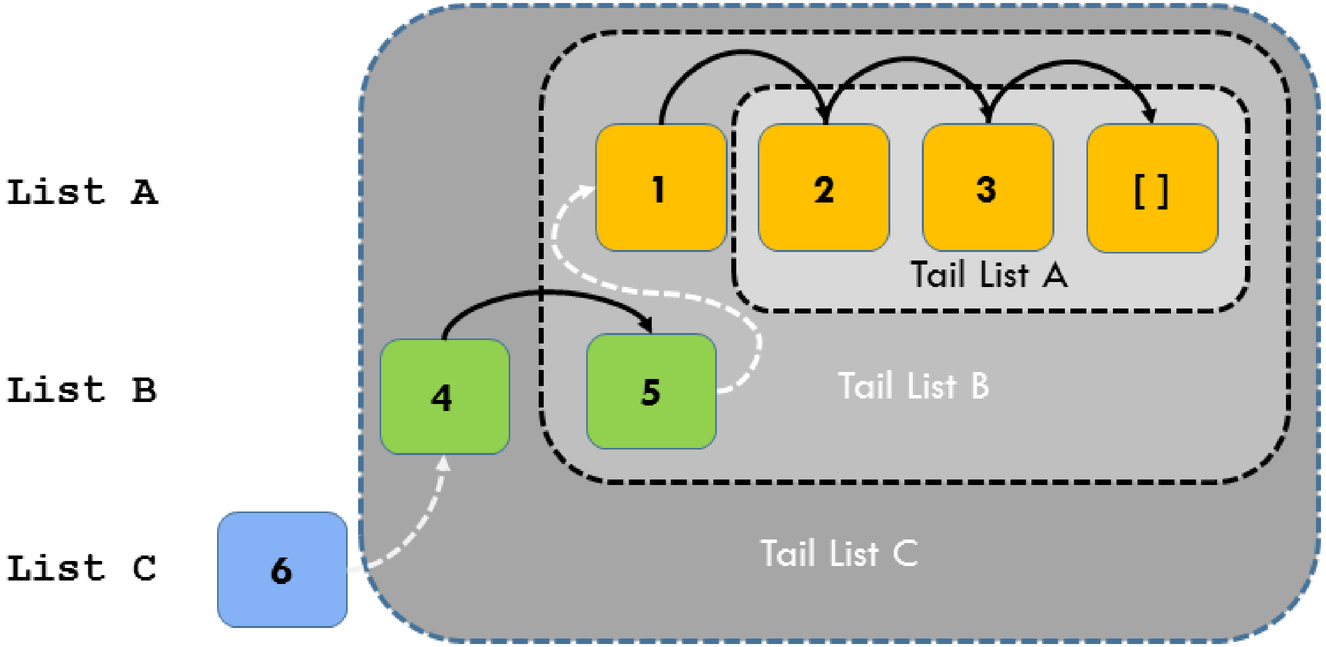
Module 4

FUNCTIONAL LISTS | UNITS OF MEASURE
| OBJECT EXPRESSION | OBJECT-ORIENTED PROGRAMMING

Functional Lists



Structural Sharing



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 ]
```

Creating a List

- **From a range expression**
  - `let integers = [1..1000]`
- **From a list expression**
  - `let integers = [for i in 1..1000 do yield i]`
  - `let integers = [for i in 1..1000 -> i]`
- **Using a function in the List module**
  - `let integers = List.init 1000 (fun i -> i+1)`
- **From another other collection**

```
let Files (dir : string) =
    Directory.EnumerateFiles(dir)
    |> List.ofSeq
```

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)
```

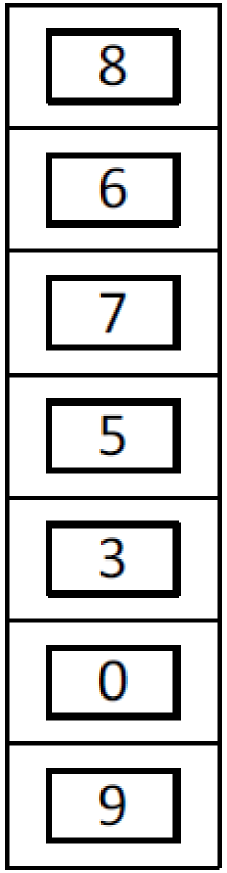
```
let vipNames = customers
                |> Seq.filter (fun c -> c.IsVip)
                |> Seq.map (fun c -> c.Name)
```

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

F#	LINQ
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

What is an Array?

- **Standard .NET type**
- **Length fixed on creation**
- **All elements of same type**
- **Array as a whole is immutable**
  - `let myArray = [|8;6;7;5;3;0;9|]`
  - `myArray <- [|8;7;7;5;3;0;9|]`
- **Elements mutable**
  - `myArray.[1] <- 7`



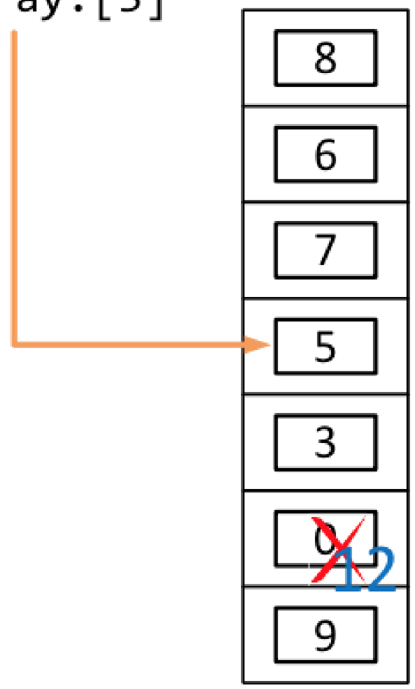
Creating an Array

- **From a literal**
  - `let primes = [|1; 3; 5; 7; 11|]`
- **From a comprehension**
  - `let someNumbers = [|1000..1020|]`
  -
    ```
    let smallEvens = [|for i in 1..100 do
                          if i%2 = 0 then yield i|]
    ```
- **Using a function from the Array module**
  - Array.create
  - Array.init
- **With zero-valued elements**
  - Array.zeroCreate
- **From another array or IEnumerable**

Accessing Array Elements

- **.[index] notation**
  - let myValue = myArray.[3]
- **Update elements with <-**

```
myArray.[3]
```




```
myArray.[5] <- 12
```

Dictionary

- **Generic mapping from keys to values**
- **Create:**
  - `let capitals = new Dictionary<string,string>()`
- **Add values:**
  - `capitals.Add("United Kingdom","London")`
  - `capitals.Add("France", "Paris")`
- **Access values:**
  - `printfn "The capital of France is %s" capitals.["France"]`

Adding or Assigning

- **Assigning using <- to the indexed value…**
  - `capitals.["Spain"] <- "Madrid"`
- **Adds if the value doesn’t pre-exist**
- **Or updates if the value does pre-exist**

Dict - bulit in Dictionary in F#

- **Create and populate in one**
- **Never in an invalid state**
- **Use 'dict'**
  - `let dictionary = dict myValues`
  - `let dictionary = myValues |> dict`
- **Input must consist of tuples**

```
let capitals =
      [
         "United Kingdom", "London"
         "United States of America","Washington D.C."
         "France", "Paris"
      ] |> dict
```

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>

Object Oriented Programming

Immutable Fields

```
type MyClass(myField: int) =

   member this.MyProperty = myField

   member this.MyMethod methodParam =
      myField + methodParam

 member static MyOtherMethod a =
      myField + a
```

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

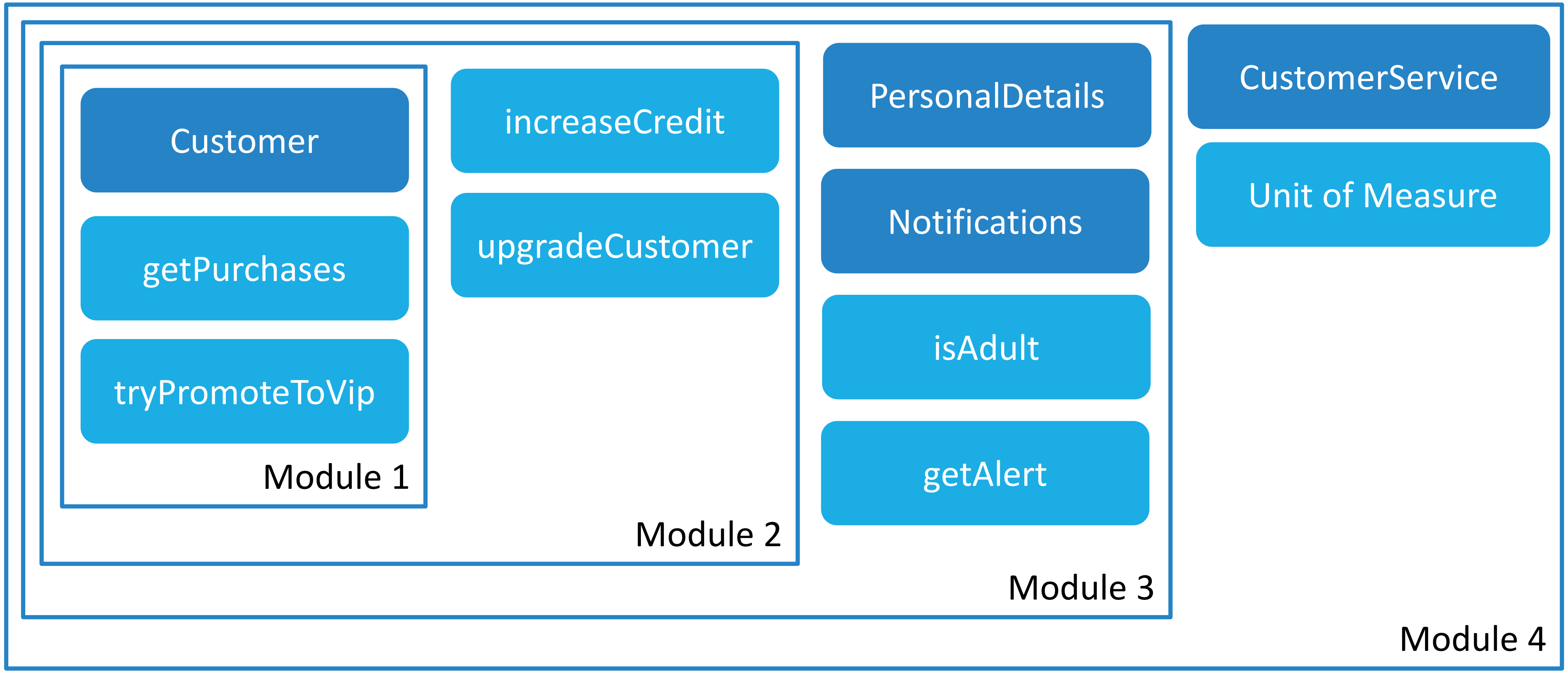
```
type IMyInterface =
   abstract member MyMethod: int -> int
```

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

Demo 4

FUNCTIONAL LISTS | UNITS OF MEASURE | OBJECT EXPRESSION | OBJECT-ORIENTED PROGRAMMING

Exercise 4


Customer
getPurchases
tryPromoteToVip
Module 1
increaseCredit
upgradeCustomer
Module 2
PersonalDetails
Notifications
isAdult
getAlert
Module 3
CustomerService
Unit of Measure
Module 4

Exercise 4

FUNCTIONAL LISTS | UNITS OF MEASURE
| OBJECT EXPRESSION | OBJECT-ORIENTED PROGRAMMING

Module 5

RECURSION | ACTIVE PATTERNS | ROP

Imperative loops

```
for item in data do
   printfn "Item value %A" item
```

```
for i = 0 to 50 do
   let item = data.[i]
   printfn "Item value %A" item
```

```
let mutable index = 0

while index < data.Length - 1 do
   let item = data.[index]
   printfn "Item value %A" item
   index <- index + 1
```

Recursion

```
let rec factorial number =
    if number = 0 then 1
    else
        printfn "Number %d" number
        number * factorial (number -
1)
```

```
let tailRecFactorial number =
    let rec fact number acc =
        if number = 0 then acc
        else
            printfn "Number %d" number
            fact (number - 1) acc * number
    fact number 1
```

Active Patterns

```
// create an active pattern
let (|Int|_|) str =
  match System.Int32.TryParse(str:string) with
  | (true,int) -> Some(int)
  | _ -> None

// create an active pattern
let (|Bool|_|) str =
  match System.Boolean.TryParse(str:string) with
  | (true,bool) -> Some(bool)
  | _ -> None
```

```
// create a function to call the patterns
let testParse str =
    match str with
    | Int i -> printfn "The value is an int '%i'" i
    | Bool b -> printfn "The value is a bool '%b'" b
    | _ -> printfn "The value '%s' is something else" str

// test
testParse "12"
testParse "true"
testParse "abc"
```

Active Patterns

```
let (|Long|Medium|Short|) (value:string) =
    if value.Length < 5 then Short
    elif value.Length < 10 then Medium
    else Long

let test () =
    match "Hello" with
    | Short -> "This is a short string!"
    | Medium -> "This is a medium string!"
    | Long -> "This is a long string!"
```

Railway Oriented Programming (ROP)

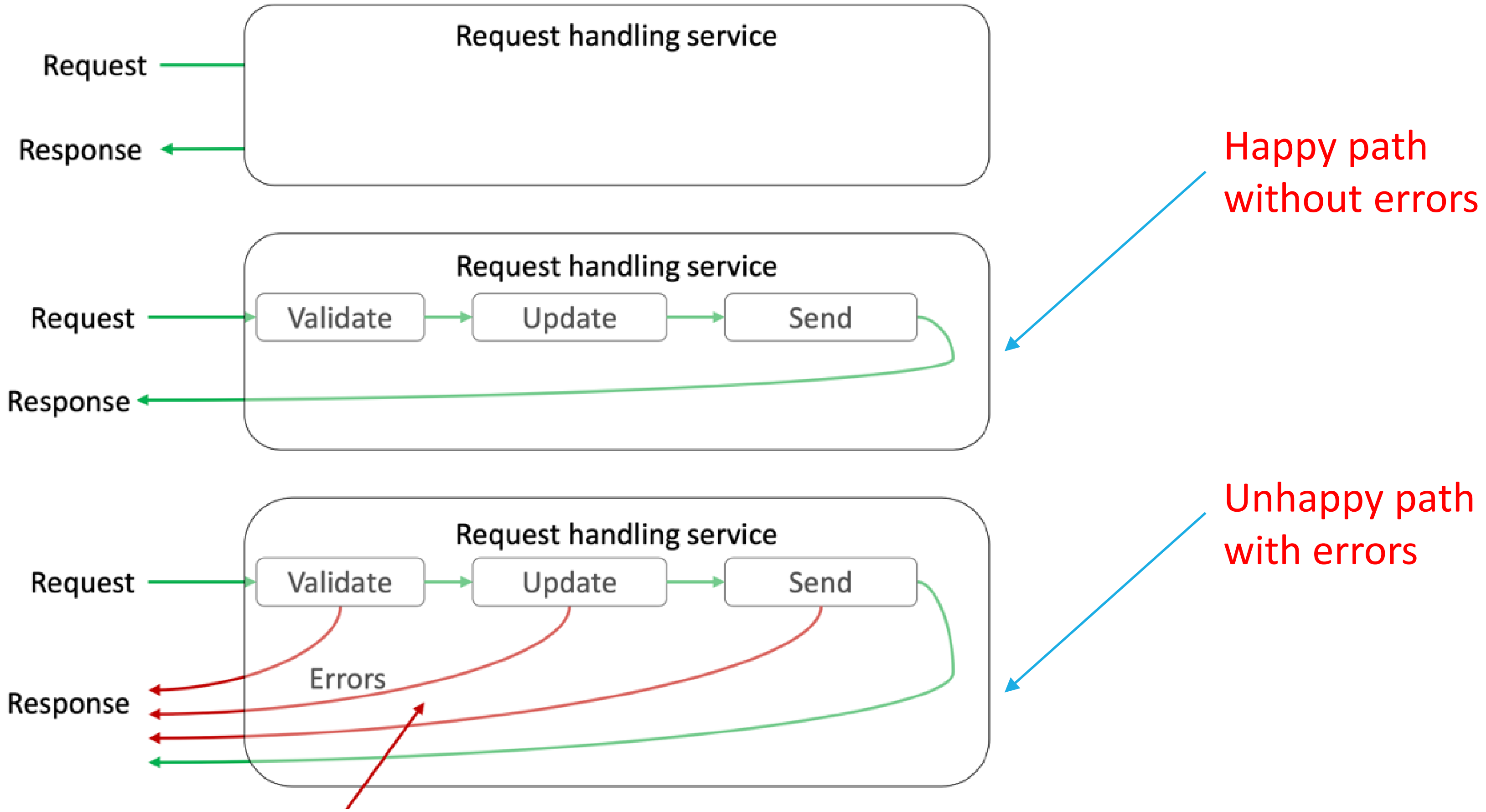
Imperative defensive programming

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  validateRequest(request);
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

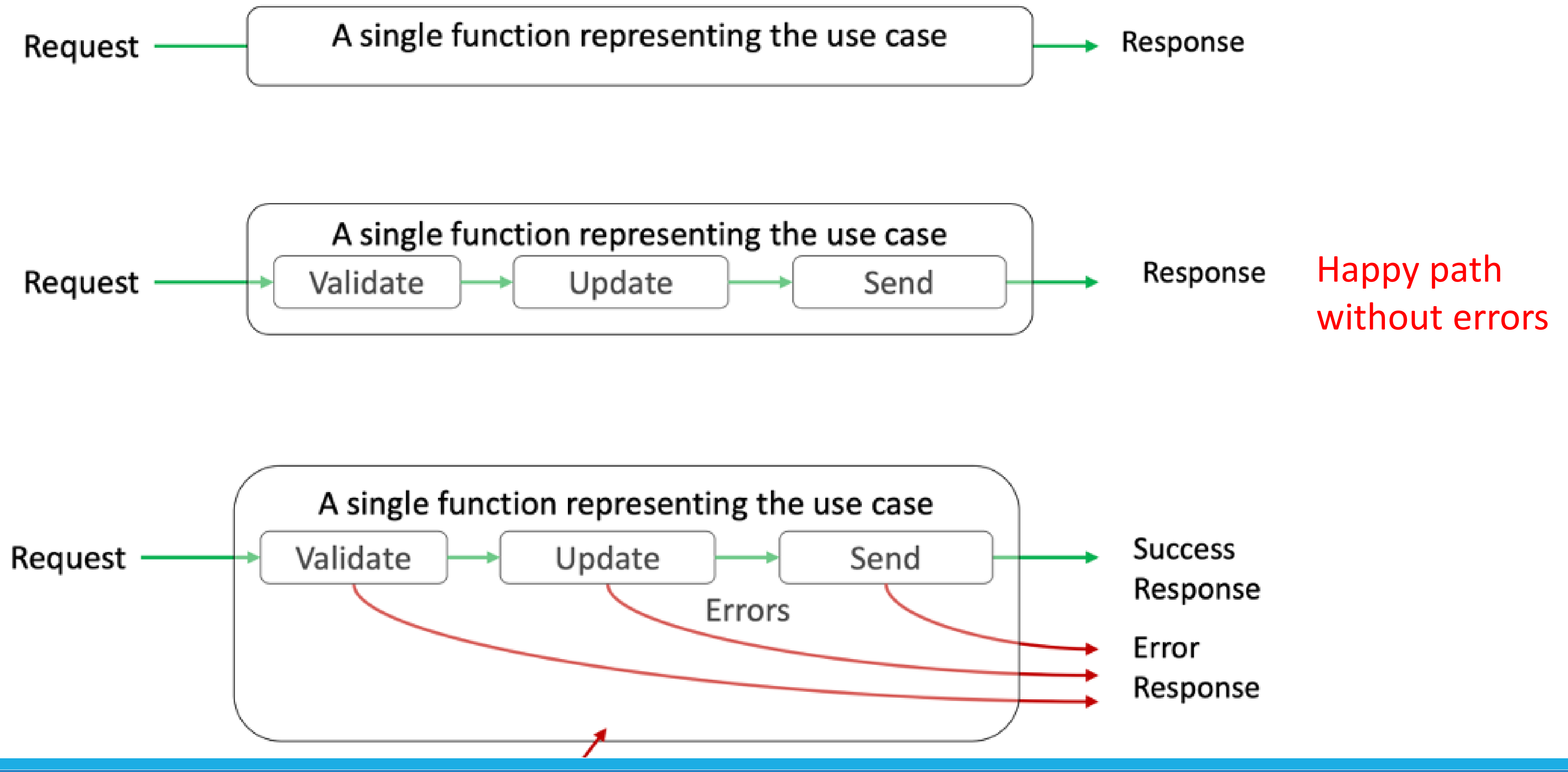
```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  var result = db.updateDbFromRequest(request);
  if (!result) {
    return "Customer record not found"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Imperative defensive programming



ROP – a functional approach to error handling

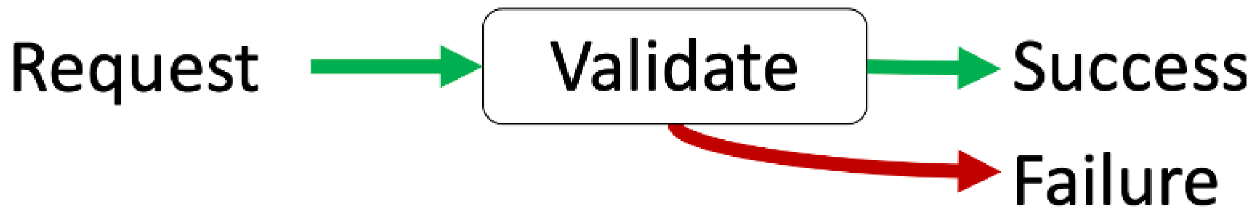


Pass multiple output downstream

ROP – a functional approach to error handling

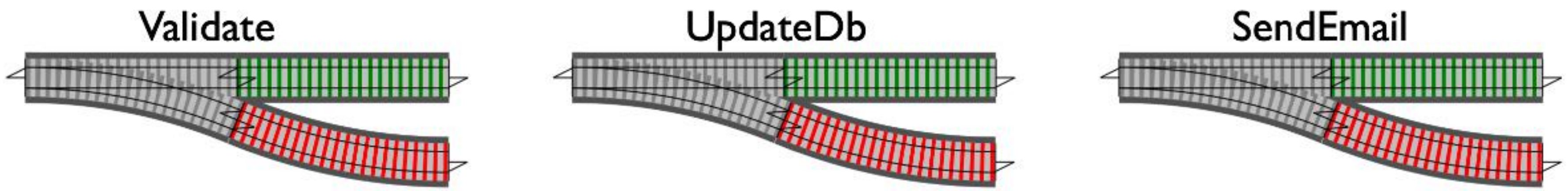
```
type Result<'TEntity> =
        | Success of 'TEntity
        | Failure of     | ValidationError
                         | UpdateError
                         | SmtpError
```

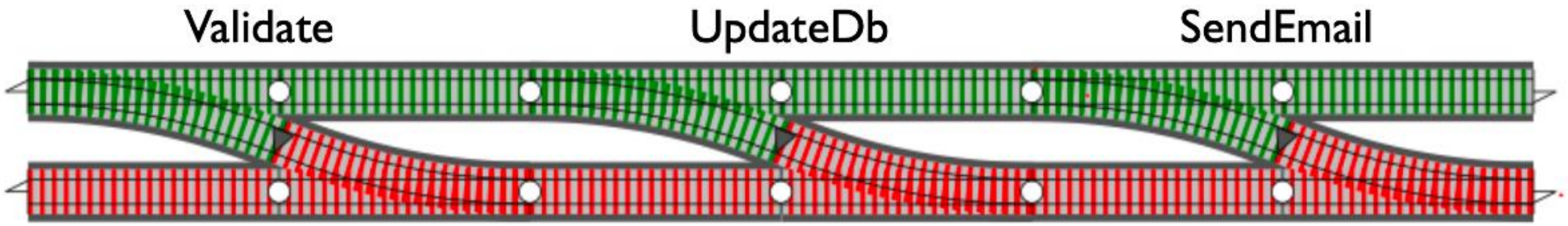
ROP – a functional approach to error handling




```
let validateInput input =
    if input.name = "" then
        Failure "Name must not be blank"
    else if input.email = "" then
        Failure "Email must not be blank"
    else
        Success input  // happy path
```

ROP – a functional approach to error handling





ROP – a functional approach to error handling

```
Result.bind : Result<T> -> (T -> Result<U>) -> Result<U>

Result.map : Result<T> -> (T -> U) -> Result<U>
```

Demo 5

RECURSION | ACTIVE PATTERNS | ROP

Exercise 5

RECURSION | ACTIVE PATTERNS | ROP

F# Koans Workshop

Module 6

DSL | ASYNC PROGRAMMING | MAILBOXPROCCESOR

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

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
```

Concurrency

Concurrency Core Concepts

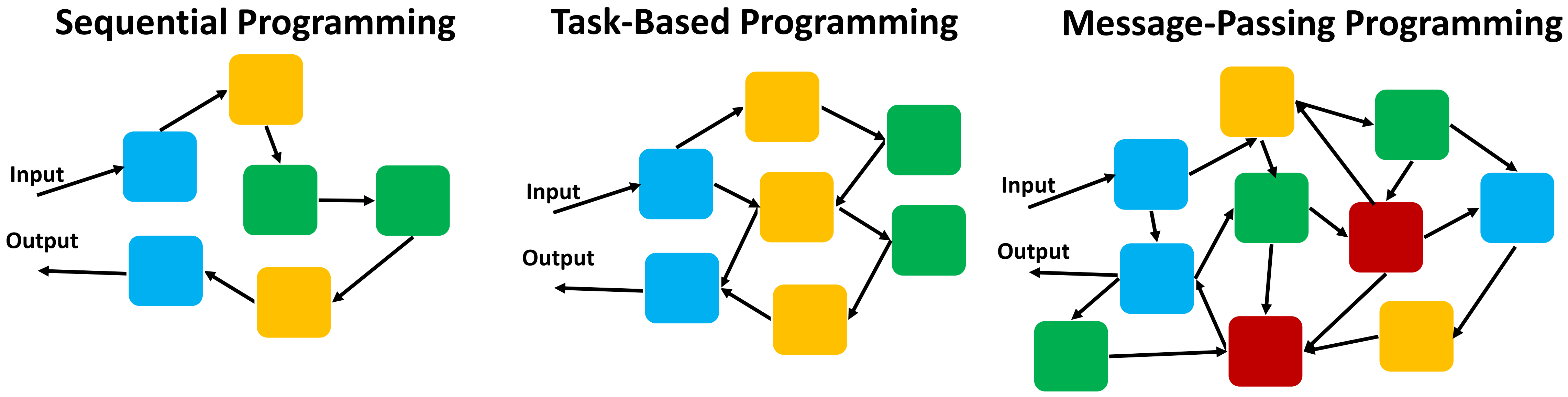
Message Passing

Reactive Programming

Data & Task Parallelism

Asynchronous Programming

Different type of concurrency models lead to different challenges

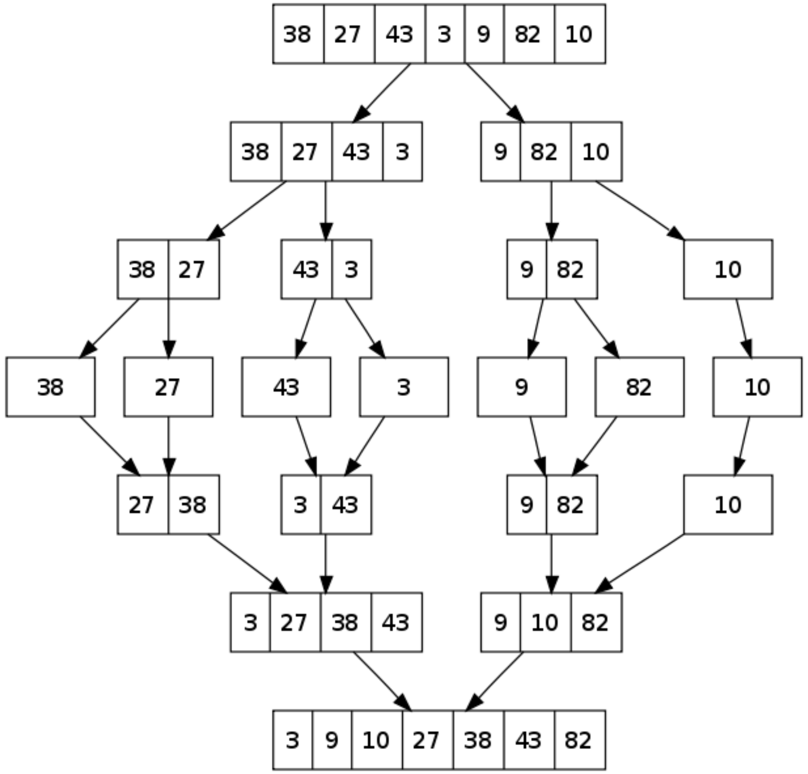


Its about maximizing resource use

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



Asynchronous Workflows

- Software is often I/O-bound, it provides notable performance benefits
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disks
- Not easy to predict when the operation will complete (non-deterministic)
- **IO bound functions can scale regardless of threads**
  - **IO bound computations can often “overlap”**
  - **This can even work a for huge numbers of computations**

(A)synchronous code

```
var wc = new WebClient();
var html = await wc.DownloadDataTaskAsync(url);
await outputStream.WriteAsync()
```

Easy to **change**, easy to **write**

Can use **loops** and **exception handling**

Scalable – no blocking of threads

Async Workflow

```
let printThenSleepThenPrint = async {
        printfn "before sleep"
        do! Async.Sleep 5000
        printfn "wake up"
    }

Async.StartImmediate printThenSleepThenPrint
printfn "continuing"
```

Async Workflow

let!	like **await** on Task<T> in C#
do!	like **await** on Task in C#
return!	like **return await** on Task<T> C#

Its all about Scalability

```
let httpAsync (url : string) = async {
        let req = WebRequest.Create(url)
        let! resp = req.AsyncGetResponse()
        use stream = resp.GetResponseStream()
        use reader = new StreamReader(stream)
        return! reader.ReadToEndAsync() }

let sites =
    [ "http://www.live.com";"  “http://www.fsharp.org";
      "http://news.live.com";  "http://www.digg.com";
      "http://www.yahoo.com";  "http://www.amazon.com"
      "http://www.google.com";     "http://www.netflix.com";
      "http://www.facebook.com"; "http://www.docs.google.com";
      "http://www.youtube.com";    "http://www.gmail.com";
      "http://www.reddit.com";     "http://www.twitter.com"; ]

sites
|> Seq.map httpAsync
|> Async.Parallel
|> Async.Start
```

Anatomy of Async Workflows

```
let readData path : Async<byte[]> = async {
  let stream = File.OpenRead(path)
  let! data = stream.AsyncRead(stream.Length)
    return data }
```

- Async defines a block of code which execute on demand
- Easy to compose

Unbounded parallelism Async.Parallel (and Async.Start)

```
let httpAsync (url : string) = async {
   let req = WebRequest.Create(url)
   let! resp = req.AsyncGetResponse()
   use stream = resp.GetResponseStream()
   use reader = new StreamReader(stream)
   return! reader.ReadToEndAsync() }

let sites =
   [   "http://www.yahoo.com";    "http://www.amazon.com"
       "http://www.google.com";   "http://www.netflix.com";
       "http://www.facebook.com"; "http://www.docs.google.com";
       "http://www.youtube.com";  "http://www.gmail.com";
       "http://www.reddit.com";   "http://www.twitter.com"; ]

|> Seq.map httpAsync
|> Async.Parallel
```

Declarative parallelism

Run in parallel and wait for completion

```
var docs = await Task.WhenAll
  (from url in pages select DownloadPage(url));
```

Functional approach

- Works nicely with F# sequences and LINQ

```
let! docs =
 Async.Parallel [ for url in urls -> downloadPage url ]
```

Async.Catch

```
let asyncTask = async { raise <|
                         new System.Exception(”My Error!") }

asyncTask
|> Async.Catch
|> Async.RunSynchronously
|> function
    | Choice1Of2 result      ->
            printfn "Async operation completed: %A" result
    | Choice2Of2 (ex : exn) ->
            printfn "Exception thrown: %s" ex.Message
```

Simple agent in F#

Receive message and say "Hello"

```
let agent = Agent.Start(fun agent -> async {
  while true do
    let! name = agent.Receive()
    printfn "Hello %s"  name
    do! Async.Sleep 500 })

agent.Post("World!")
```

- Single instance of the body is running
- Waiting for message is asynchronous
- Messages are queued by the agent

Simple agent with state in F#

```
type Message =
   | Add of string
   | GetNames of AsyncReplyChannel<string list>

let agent = Agent.Start(fun agent ->
  let rec loop names = async {
    let! msg = agent.Receive()
    match msg with
    | Add name -> return! loop (name::names)
    | GetNames channel -> channel.Reply(names)
                                            return! loop names }
  loop [])

agent.Post(Add "Bella")
agent.Post(Add "Stellina")

let names = agent.PostAndReply(fun ch -> GetNames ch)
for name in names do
    printfn "Name is %s" name
```

Mutable and immutable state

Mutable state

- Accessed from the body
- Used in loops or recursion
- Mutable variables (ref)
- Fast mutable collections

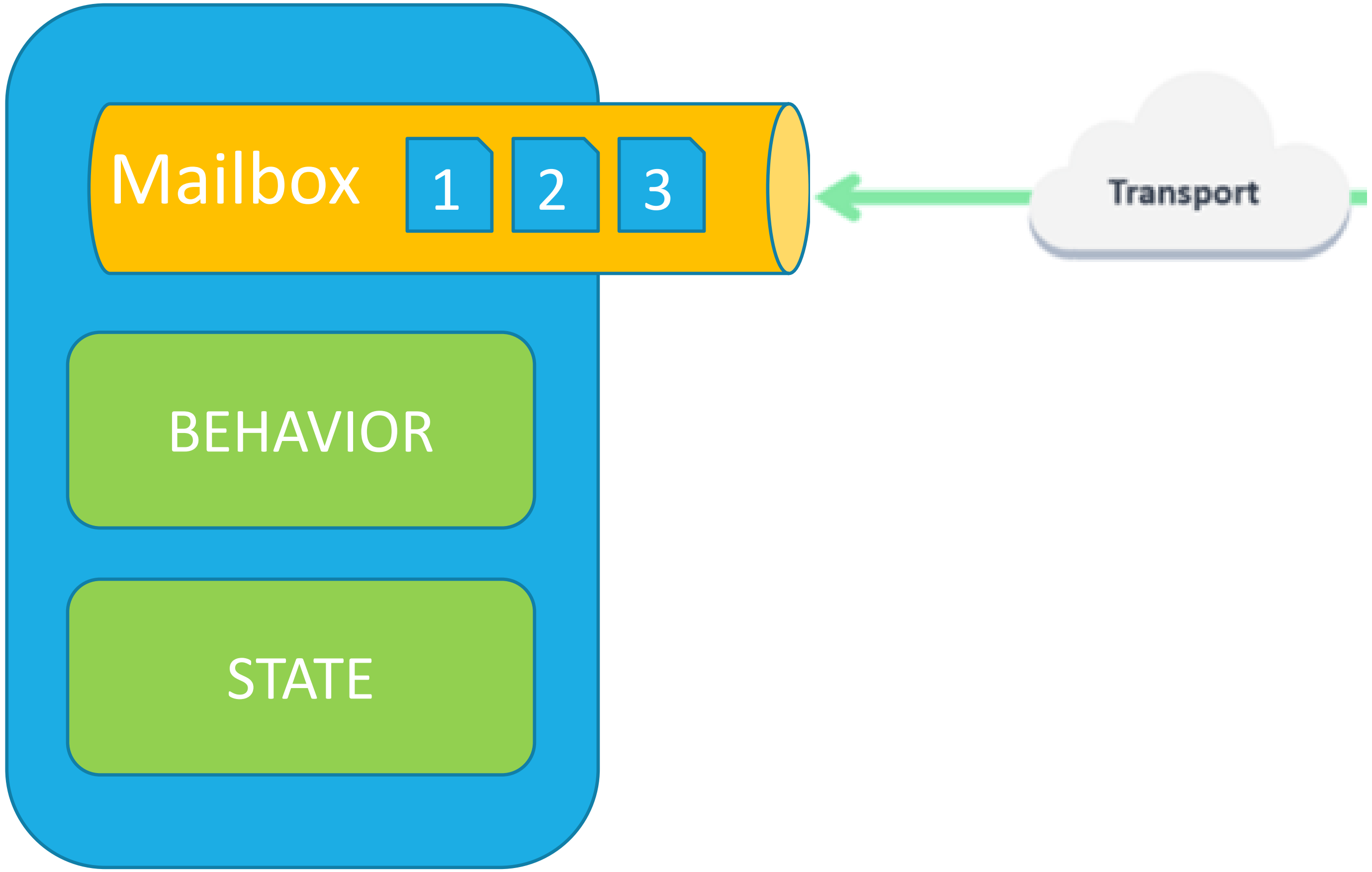
Immutable state

- Passed as an argument
- Using recursion (**return!**)
- Immutable types
- Can be returned from the agent

```
Agent.Start(fun agent -> async {
  let names = ResizeArray<_>()
  while true do
    let! name = agent.Receive()
    names.Add(name) })
```

```
Agent.Start(fun agent ->
  let rec loop names = async {
    let! name = agent.Receive()
    return! loop (name::names) }
 loop [])
```

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 6

CONCURRENCY | ASYNC PROGRAMMING | AGENT

Exercise 6

PARALLEL WEB CRAWLER

Resources

fsharp.org / c4fsharp.net

Real-World Functional Programming
By Tomas Petricek

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

fsharp.tv

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

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


That's all Folks!