Reactive and Concurrent F#

& Workshop

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

— Edward A. Lee *(The Problem with Threads, Berkeley 2006)*

Riccardo Terrell – F# Gotham

Agenda

Module1

F# Async Workflow

Module2

F# Agent Model

F# & Reactive Extensions

Workshop 1

Fast Zoom in & Zoom out

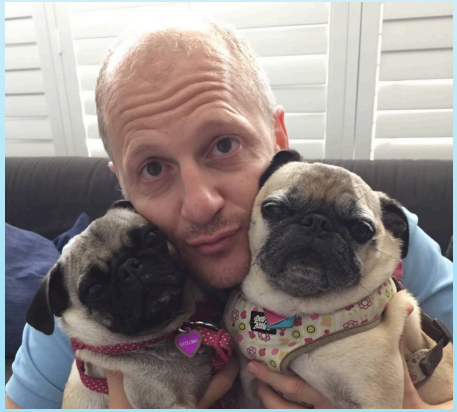
Workshop 2

Distributed Stock-Tickers

Workshop 3

Drag & Draw with undo

Something about me – *Riccardo Terrell*



- Originally from Italy, currently living in USA ~10 years
  - Living/working in Washington DC
- +/- 18 years in professional programming
  - C++/VB → Java → .Net C# → Haskell → F# → Scala → ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Believer in polyglot programming as a mechanism in finding the right tool for the job

rickyterrell.com @trikace rterrell@microsoft.com

Goals

Today developers must embrace concurrent programming

Functional Programming helps to go Reactive

F# really **shines** in the area of concurrency and distributed computing

Get motivation to practice and master F# concurrent programing model

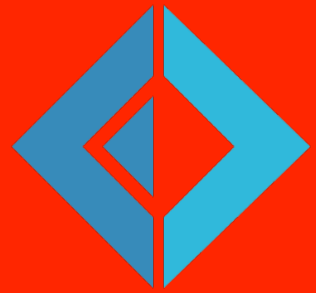
Materials

Exercises source code

- https://github.com/rikace/GothamFS

Visual Studio 2013 Professional or higher

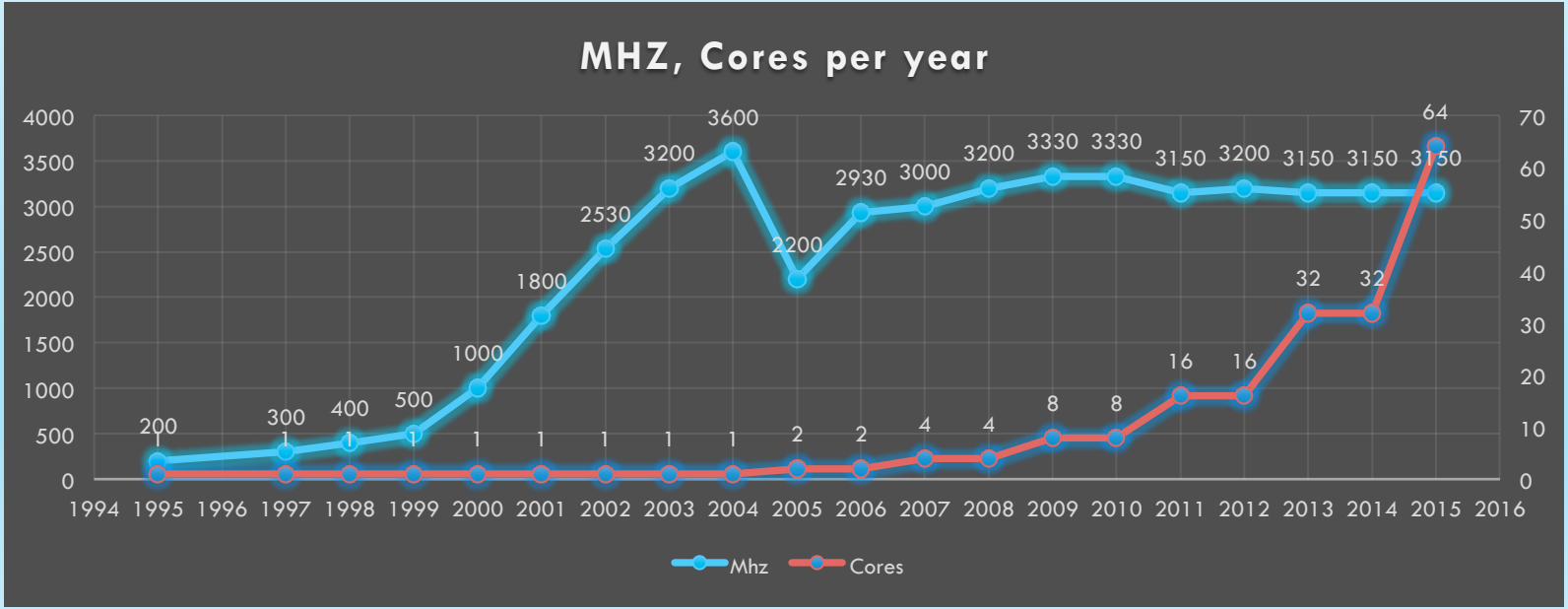
Visual F# Power Tools (optional)



Concurrency

Moore's law - The Concurrency challenge

Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!

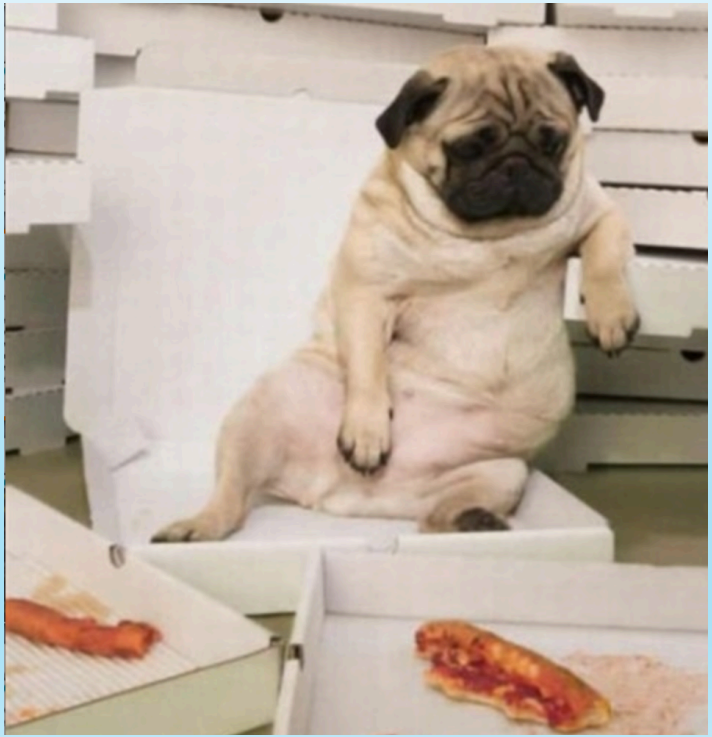


The free lunch is over

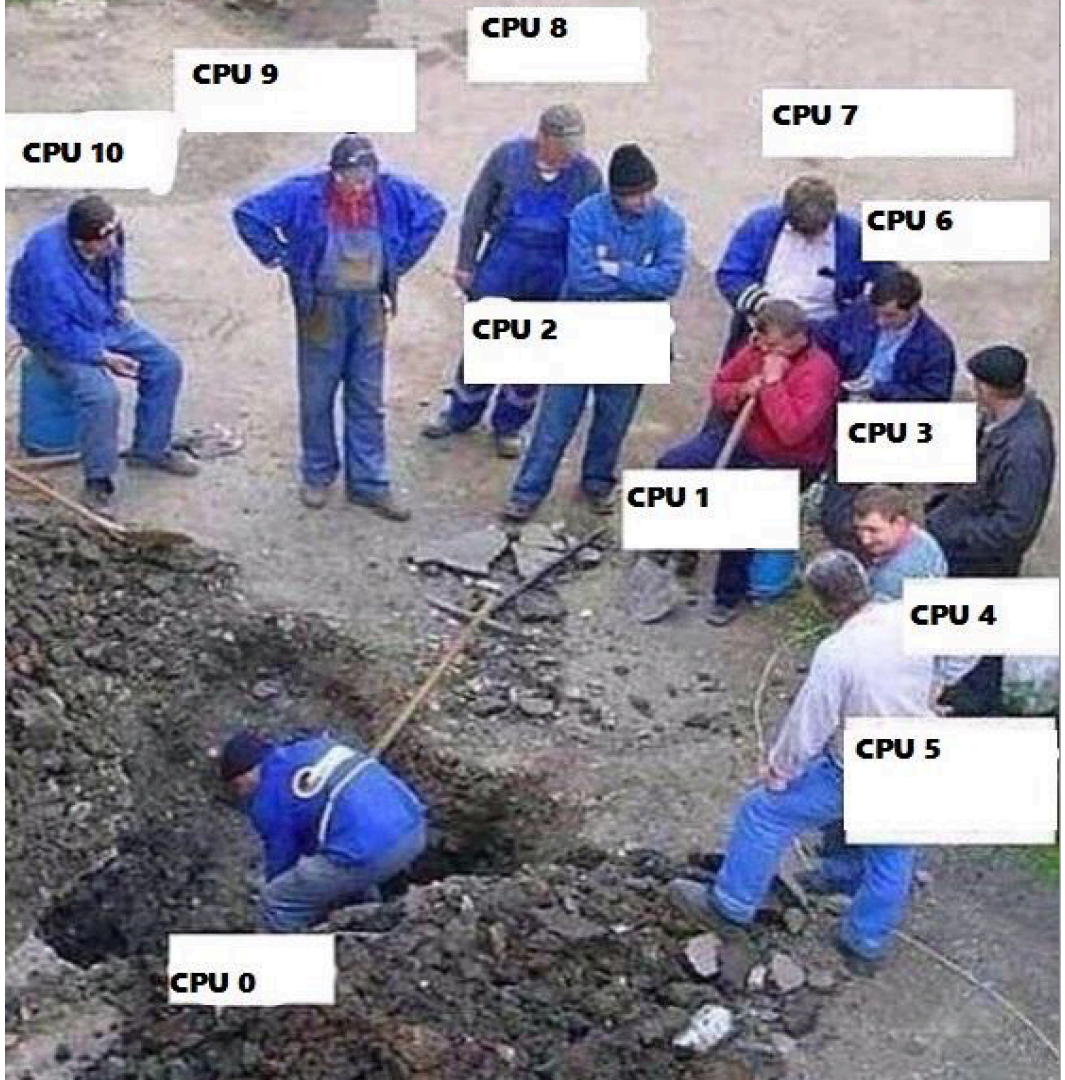
There is a problem…

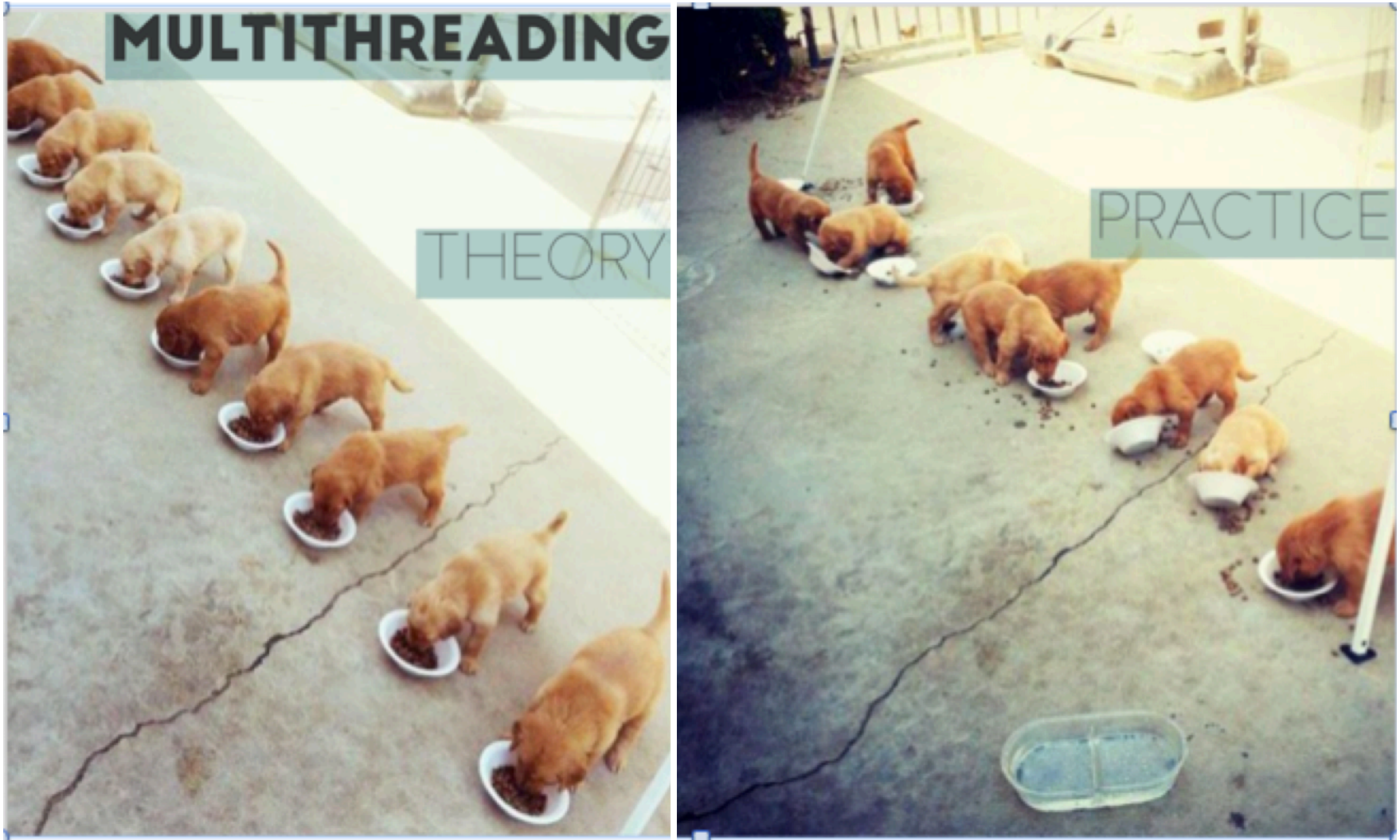
the free lunch is over

- *Programs are NOT doubling in speed every couple of years for free anymore*
- *We need to start writing code to take advantage of many cores*



https://www.cs.utexas.edu/~lin/cs380p/Free_Lunch.pdf


CPU 8
CPU 9
CPU 7
CPU 10
CPU 6
CPU 2
CPU 3
CPU 1
CPU 4
CPU 5
CPU 0


MULTITHREADING
THEORY
PRACTICE

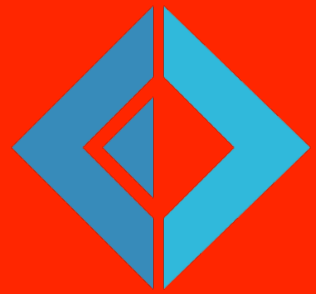

MUTABLE SHARED STATE
MUTABLE SHARED STATE EVERYWHERE

The issue is Shared of Memory

- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Not fun in a multi-threaded environment
- Not fun trying to parallelize
- Locking, blocking, call-back hell

Immutability

- *Purely functional languages do not use mutable data or mutable states, and thus are inherently thread safe – the output value of a function depends entirely on its arguments.*
  - This frees up the programmer from having to manage complex thread synchronization / locks



Module 1
Async Workflow

Asynchronous Workflows

- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasing slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)
- ***IO bound functions can scale regardless of threads***
  - ***IO bound computations can often "overlap"***
  - ***This can even work for huge numbers of computations***

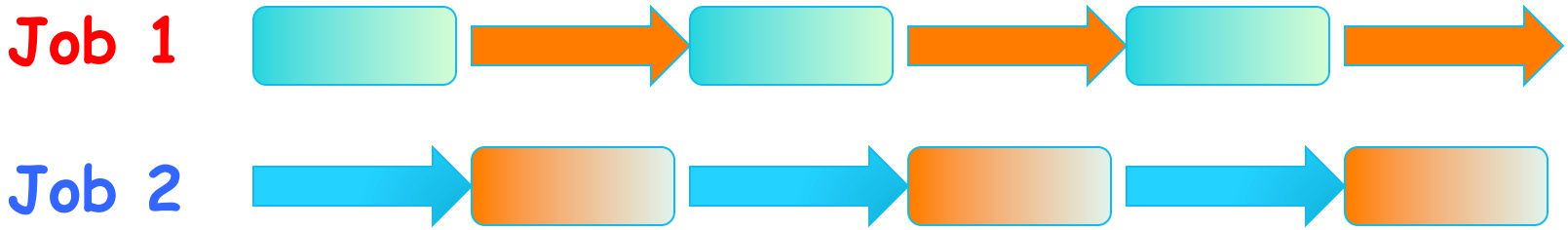
Synchronous Programming

```
let readData path =
  let stream = File.OpenRead(path)
  let data = Array.zeroCreate<byte> stream.Length
  let bytesRead = stream.Read(data, 0, data.Length)
```

- Blocks thread while waiting
  - Does not scale
  - Blocking user interface – when run on GUI thread
  - Simple to write – loops, exception handling etc

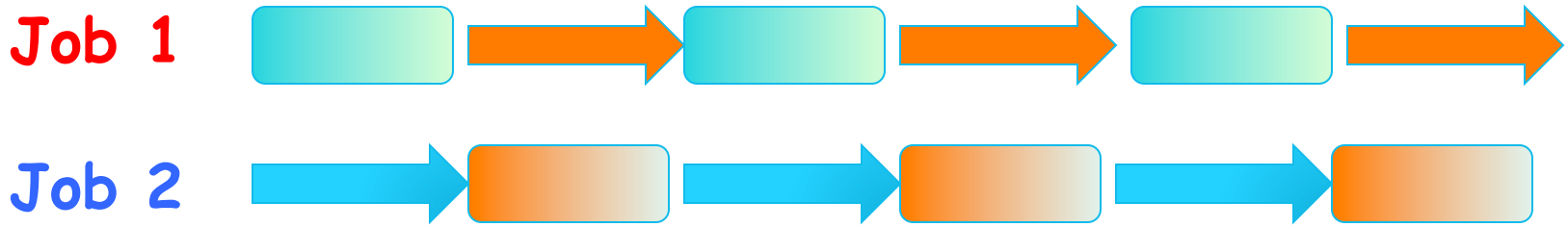
Classic Synchronous programming

- We are used to writing code linearly

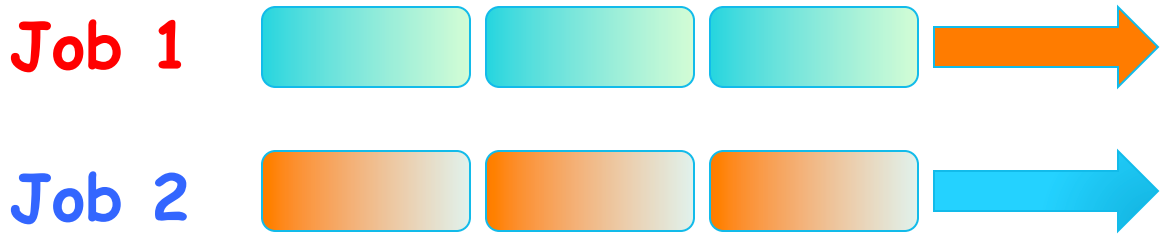


Classic Synchronous programming

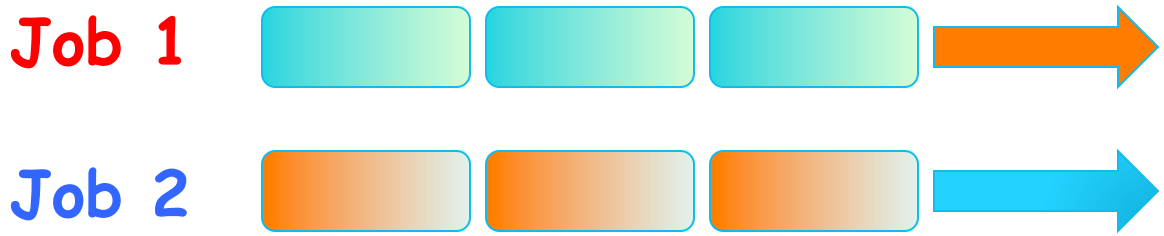
- We are used to writing code linearly



- Jobs executing in parallel



Classic Asynchronous programming



- Classic Asynchronous programming requires decoupling Begin from End
- Very difficult to:
  - Combine multiple asynchronous operations
  - Deal with exceptions, cancellation and transaction

Classic Asynchronous Programming

```
let callBack (callBack:I                    =
    let fs = call    As                    Stream
    fs.EndRead(ca
    fs.Dispose()
```

```
let f
    let                                    te> fs.Length
    let as
        .Begin    d(d      0, data.Length,
                                    callBack, fs)
```

Where is my DATA ?
@#!$#!$?!!

Anatomy of Async Workflows

```
let readData path = async {
  let stream = File.OpenRead(path)
  let! data = stream.AsyncRead(stream.Length)
  return data }
```

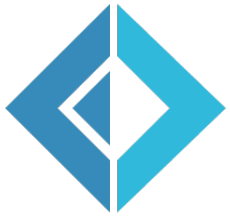
- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

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

Async Composition

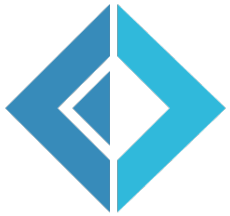


```
let readData path = async {  // string -> Async<byte[]>
      let stream = File.OpenRead(path)
      return! stream.AsyncRead(stream.Length) }
```

```
string -> byte[] -> Async<unit>
let writeData path data = async {
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination = async {
      let! data = readData source
      do!  writeData destination data }
```

Async Composition



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

```
string -> Async<byte[]> -> Async<unit>
let writeData path getData = async {
      let! data = getData
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination : Async<unit> =
   readData source |> writeData destination // Async.Start
```

Parallelizing Async

- Parallel composition of workflows (Fork-Join)

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

- Task-based parallelism (Promise-Based)

```
async {
  let! dp1 = Async.StartChild(downloadPage(url1))
  let! dp2 = Async.StartChild(downloadPage(url2))
  let! page1 = dp1
  let! page2 = dp2 }
  return (page1, page2) }
```

Exceptions & Parallel

Creates an asynchronous computation that executes all the given asynchronous computations queueing each as work items and using a fork/join pattern.

```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
  Async.Parallel[for i in 0..40 -> async { return fib(i) }]
  |> Async.Catch
  |> Async.RunSynchronously
  |> function
    | Choice1Of2 result > printfn "Successfully %A" result
    | Choice2Of2 exn -> printfn "Exception occured %s" exn.Message
```

Asynchronous Workflows - Cancel

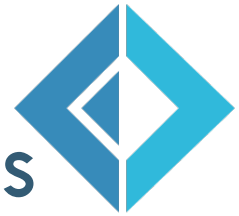
The Asynchronous workflows can be cancelled… correctly!

```
let token = new CancellationTokenSource()

Async.Start(readFileAsynchronous, token.Token)

token.Cancel()
```

Asynchronous Workflows - Continuations

```
let token = new CancellationTokenSource()
let continuation result =
        printfn "Async operation completed: %A" result
let exceptionContinutaion (ex:exn) =
        printfn "Exception thrown: %s" ex.Message
let cancellationContinuation (cancel:OperationCanceledException) =
        printfn "Async operation cancelled"

    Async.StartWithContinuations(readFileAsynchronous,
            continuation, // ‘a -> unit
            exceptionContinuation, // exn -> unit
            cancellationContinuation,  // opc -> unit
            token.Token)
```

Async Workflows for GUI interactions



`Async.AwaitObservable` operation

```
AwaitObservable : IObservable<'T> -> Async<'T>
```

```
AwaitObservable : IObservable<'T> * IObservable<'U>
                  -> Async<Choice<'T, 'U>>
```

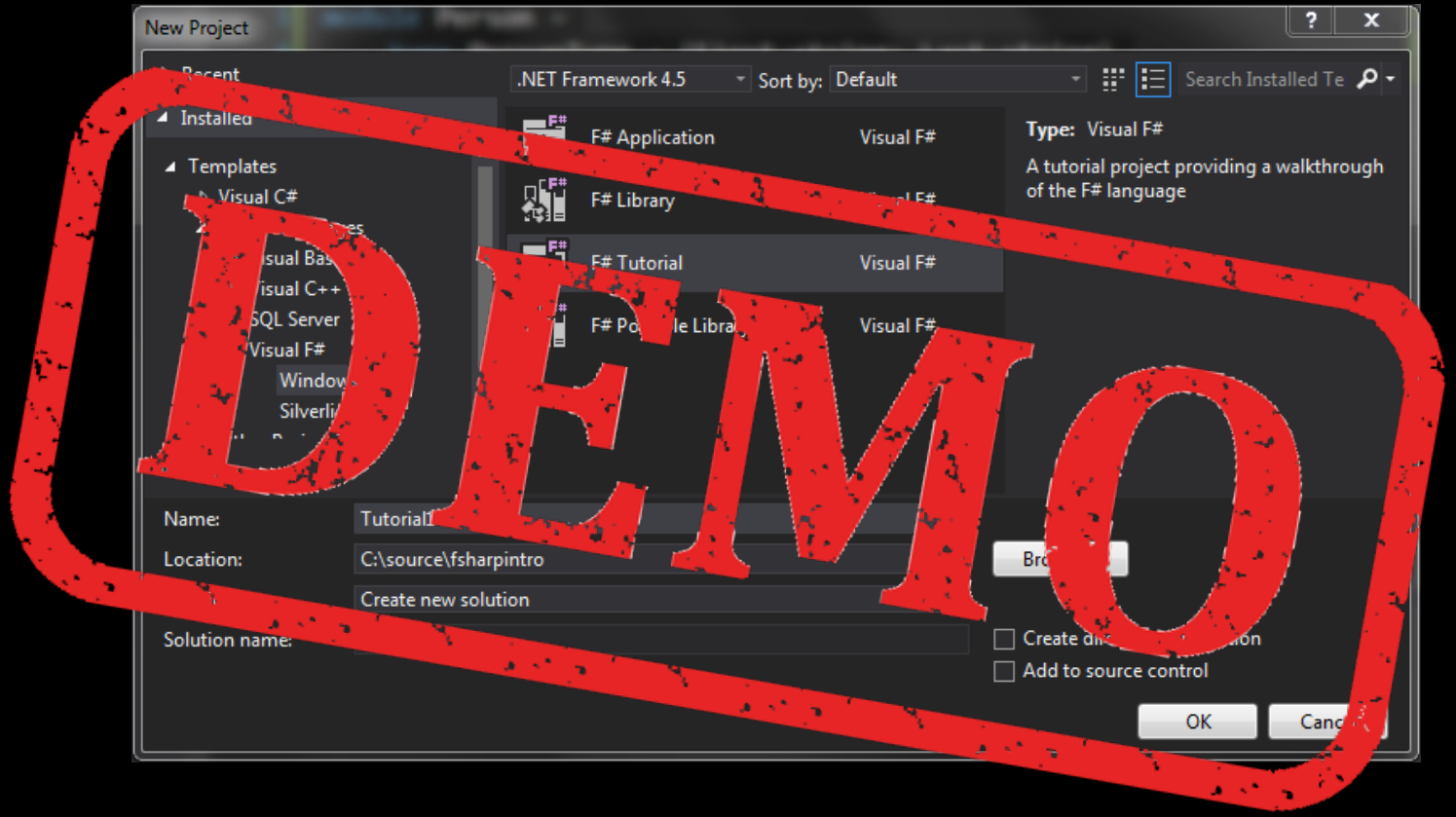
- Creates workflow that waits for the first occurrence
- Currently not part of F# libraries
- Works because `IEvent<'T> : IObservable<'T>`

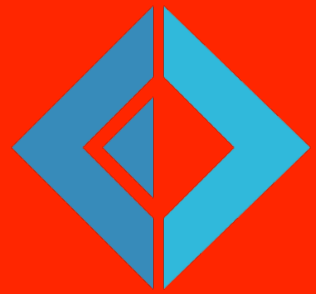
Writing loops using workflows

```
let displayMousePosition() = async {
  while true do
     let! md = Async.AwaitObservable(this.MouseMove)
     display(md.GetPosition(main) } |> Async.StartImmediate
```

Choosing between two (or more) events

```
let! evt = Async.AwaitObservable
              (this.MouseLeftButtonDown, this.MouseMove)
match evt with
| Choice1Of2(up) ->    // Left button was clicked
| Choice2Of2(move) ->  // Mouse cursor moved }
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

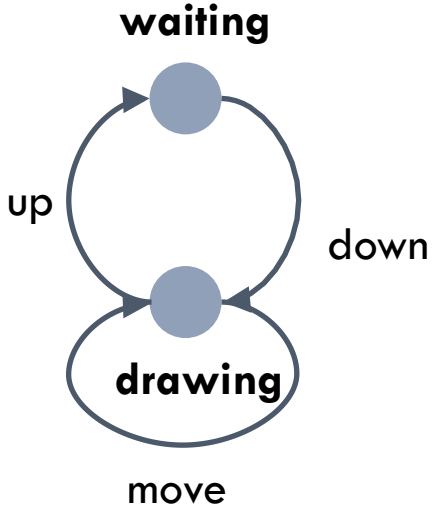


Workshop1
Async Workflow

Estimated Time ~25 mins

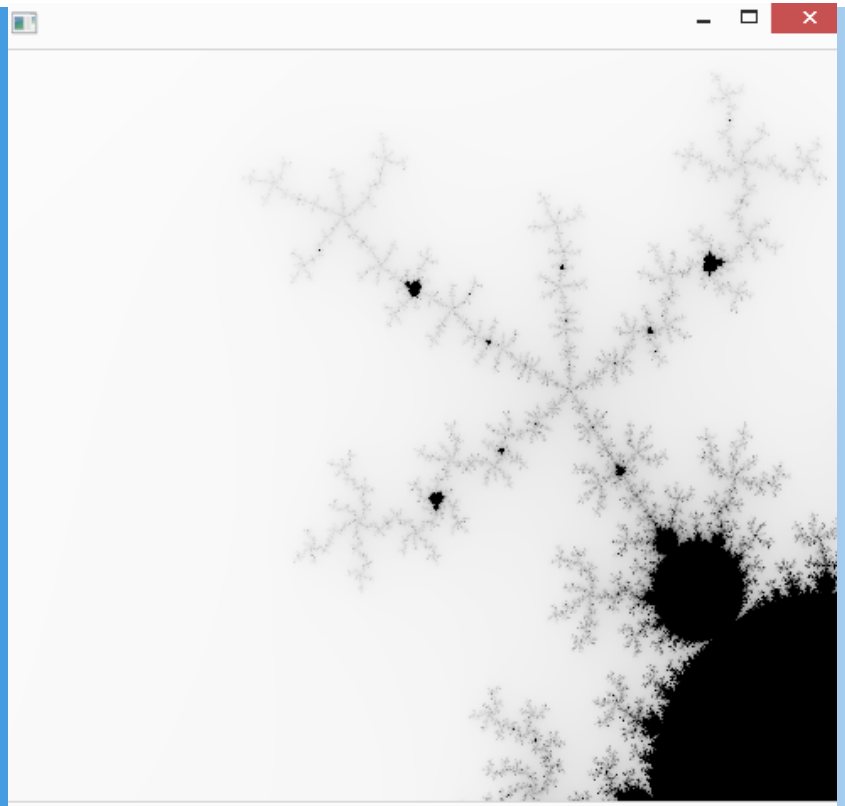
Fractal-Zoom

Fractal-Zoom



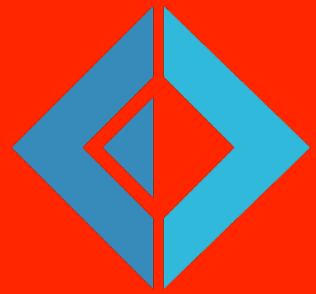

```
let rec waiting() = async {
    let! md = Async.AwaitObservable(self.MouseDown)
    // ...
   do! drawing(rc, md.GetPosition(canvas)) }
and drawing(rc:Canvas, pos) = async {
   let! evt = Async.AwaitObservable(canvas.MouseUp,
                                    canvas.MouseMove)
   match evt with
   | Choice1Of2(up) ->
         // ...
         do! waiting()
   | Choice2Of2(move) ->
         // ...
         do! drawing(rc, pos) }
do  waiting() |> Async.StartImmediate
```

Fractal-Zoom



Tasks

1. Convert current functionality to run in parallel
2. Implement Zoom Out

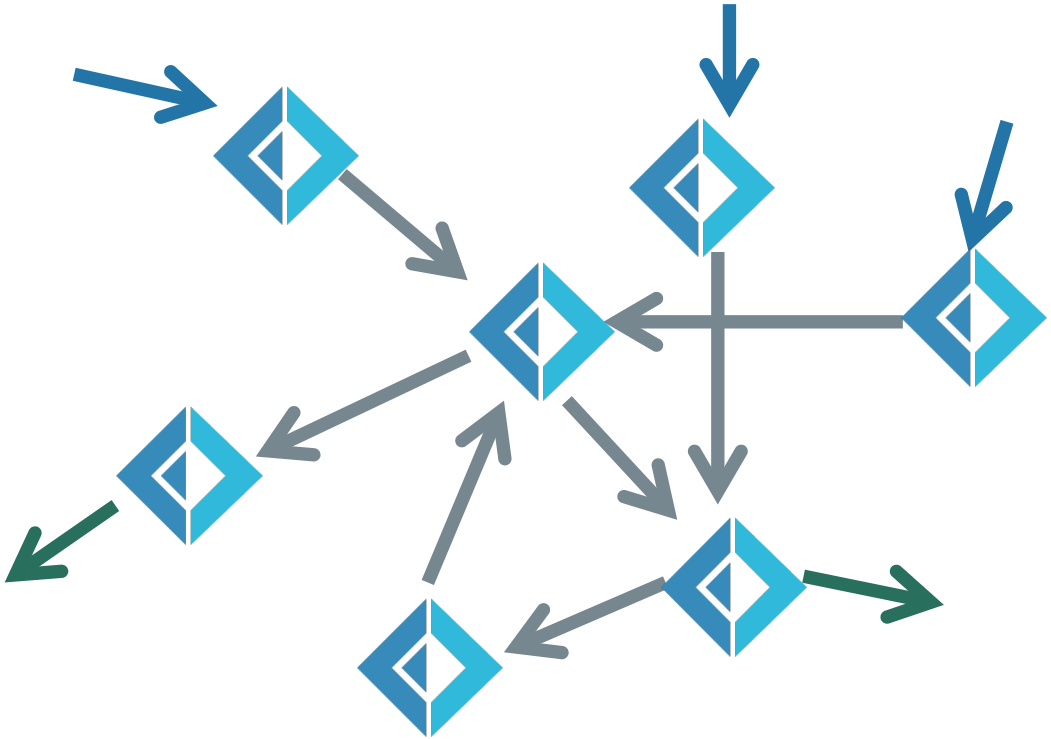


Module 2
MailboxProcessor (aka Agent)

MailboxProcessor (aka Agent)

- 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.
- Massive scalable and lightening fast because of the small call stack.

Agent-based concurrency



Agents **exchange** messages

- Receive message and react

Reactive system

- Handle inputs while running
- Emit results while running

Anatomy of an Agent

```
let agent = Agent<_>.Start(fun mb ->
                let rec loop count = async {
                      let! msg = mb.Receive()
                      match msg with
                      | Add(n) -> return! (count + n)
                      | Get(reply) -> reply.Reply(count)
                      return! loop count }
                loop 0 )
agent.Post(Add(42))
```

Message passing using F# `MailboxProcessor`

Processors react to received messages

Immutability OR Isolation

```
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] <- value
            | Pull (key,fetch) -> fetch.Reply cache.[key]
        return! loop ()
        }
    loop ())
```

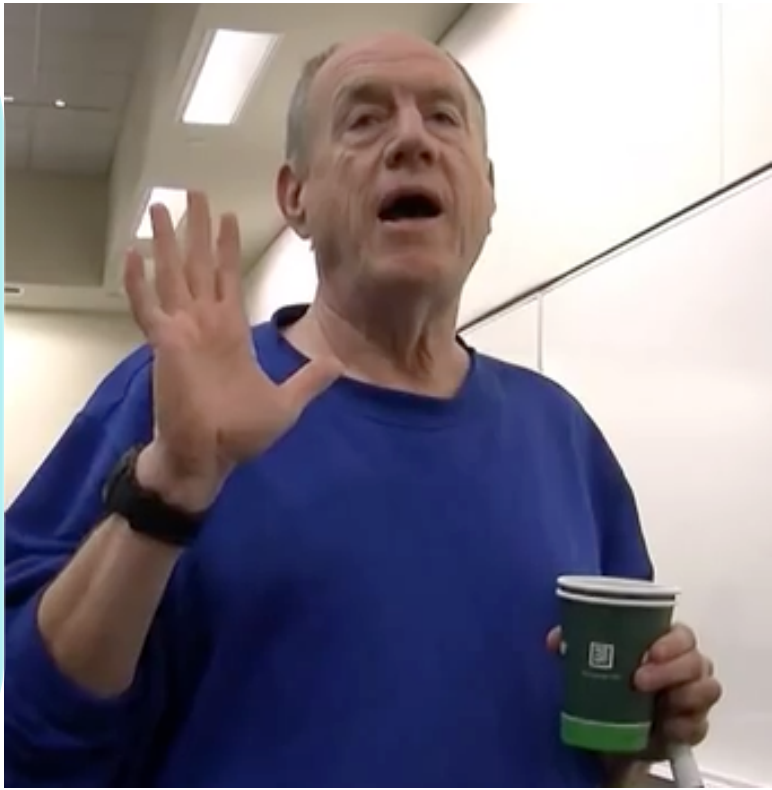
Multi-Threads vs Multi Agents

1 MB per thread (4 Mb in 64 bit) vs 2 million Agents per Gigabyte

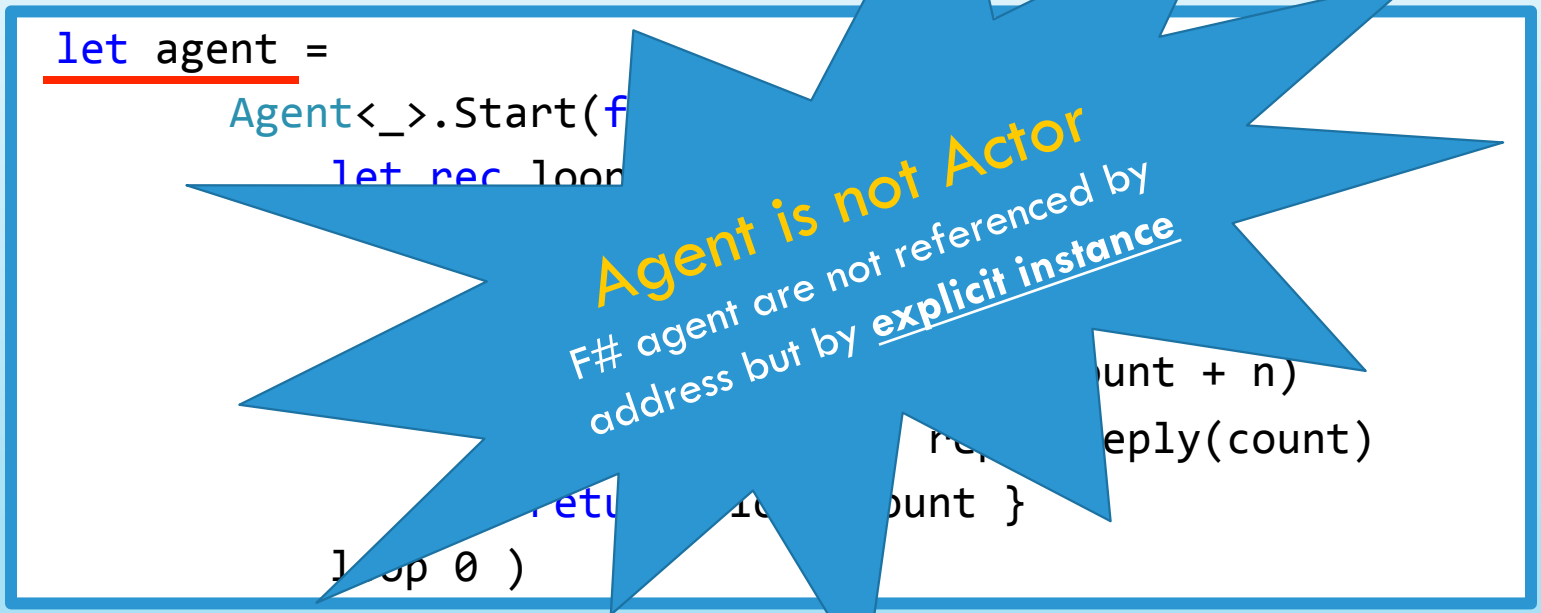
VS

Carl Hewitt's Actor Model

- **Determine** what to do with the next incoming message
- **Create** more actors
- **Send** messages to another actor



Anatomy of an Agent



Agent is not Actor

F# agent are not referenced by address but by **explicit instance**

Agent Error Handling & Disposable

```
let errorAgent =
       Agent<int * System.Exception>.Start(fun inbox ->
         async { while true do
                   let! (agentId, err) = inbox.Receive()
                   printfn "an error '%s' occurred in agent %d" err.Message agentId })

let agent cancellationToken =
         new Agent<string>((fun inbox ->
                   async { while true do
                             let! msg = inbox.Receive()
                             failwith "fail!" }), cancellationToken.Token)
          agent.Error.Add(fun error -> errorAgent.Post (error))
          agent.Start()
          agent


// (agent :> IDisposable).Dispose()
```

Agent Replying to the sender

- **Message** carries input and a callback

```
type Message = string * AsyncReplyChannel<string>
```

- **Reply** using the callback object

```
let echo = Agent<Message>.Start(fun agent ->
 async { while true do
           let! name, rchan = agent.Receive()
           rchan.Reply("Hello " + name) })
```

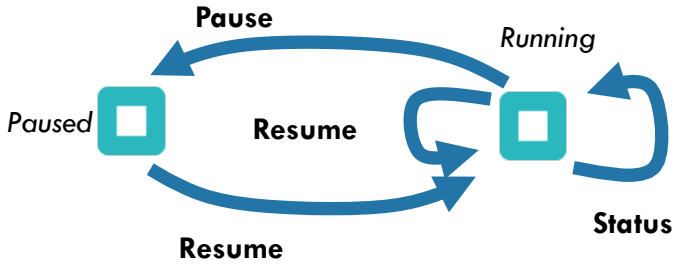
- **Asynchronous** communication

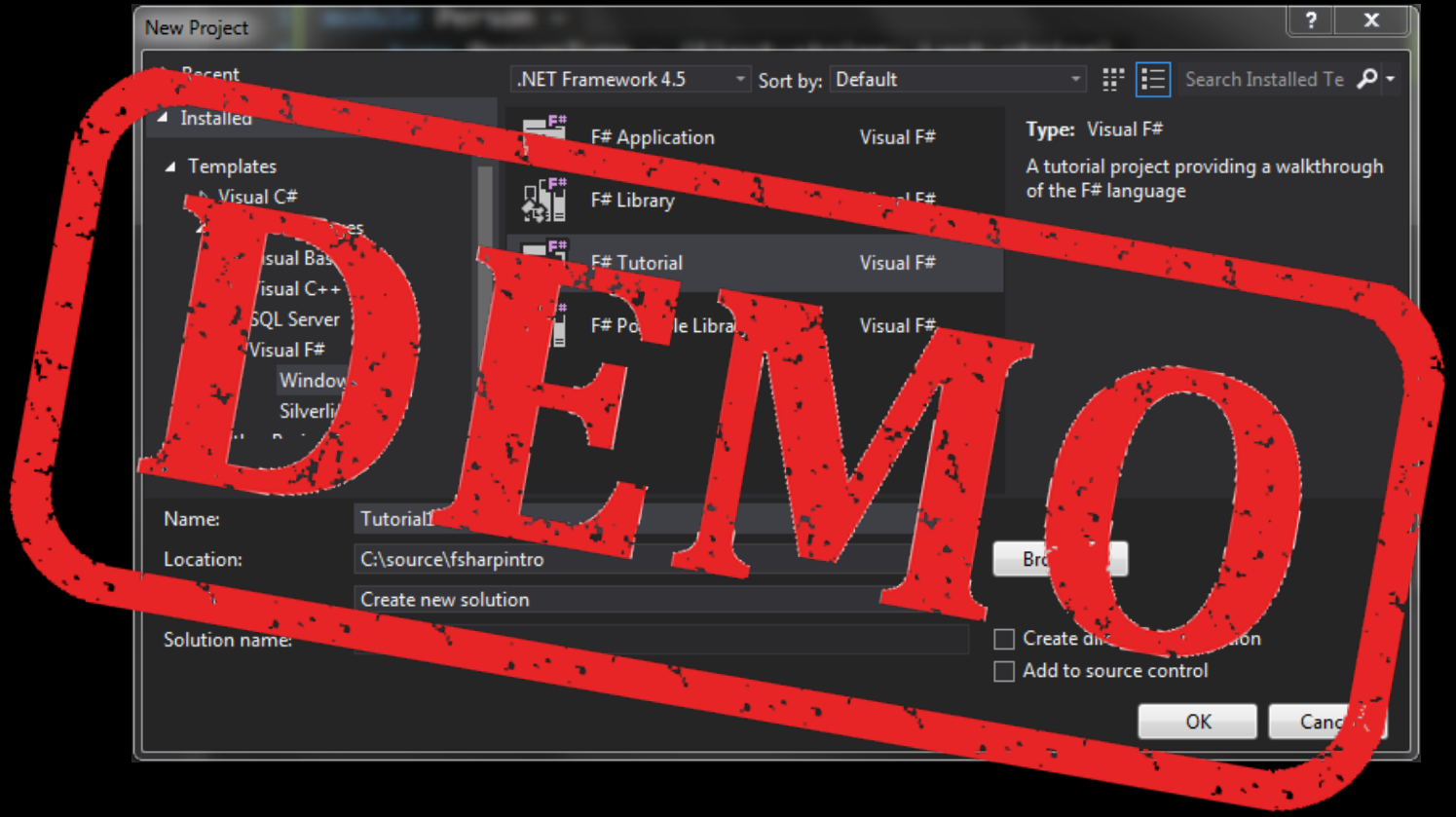
```
let! s = echo.PostAndAsyncReply(fun ch -> “F#", ch)
```

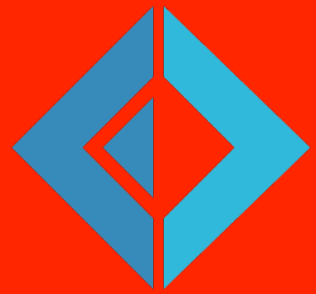
Agent State Machine

- Multi-state agents use state machines
  - Easy to implement as recursive functions
  - Some states may leave messages in the queue

```
Agent.Start(fun agent ->
  let rec paused = agent.Scan (function
    | Resume -> Some (async {
        printfn "Resumed!"
        return! running })
    | _ -> None)
  and running = (* ... *) )
```



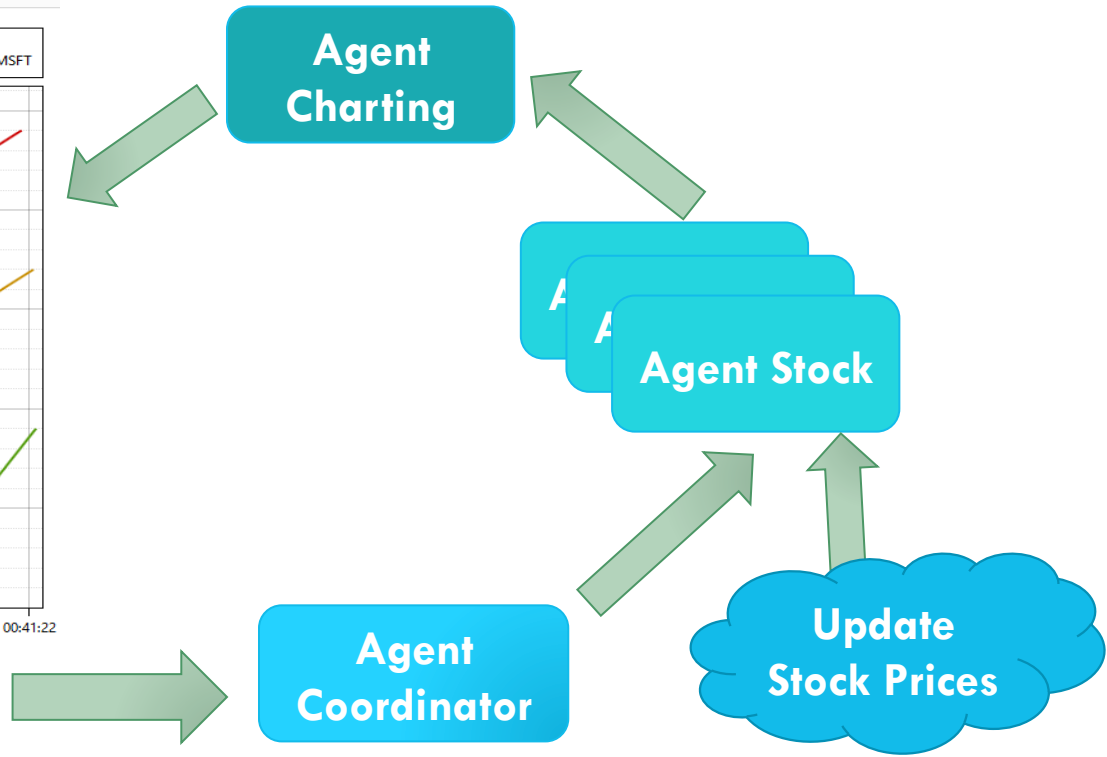

New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO



Workshop 2
MailboxProcessor (aka Agent)

Estimated Time ~20 mins

Agent Stock Ticker



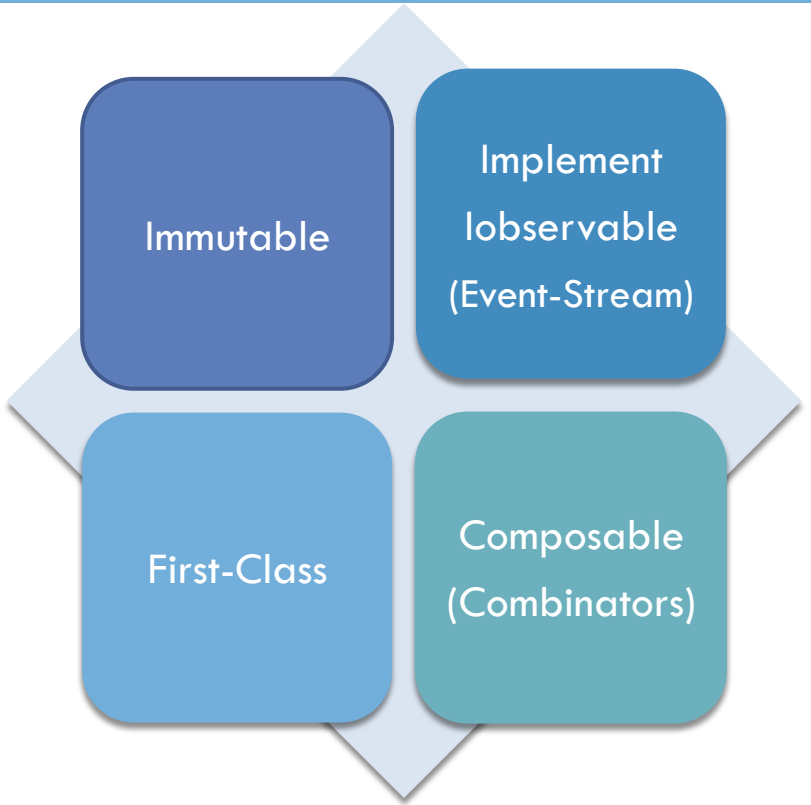
Agent Stock Ticker

Tasks

1. Create Agent hierarchy Children-Parent (sub-pub)
   1. Create Agent Stock (one per stock symbol)
   2. Create Agent Coordinator for subscribe/ unsubscribe Agent Stocks
2. Connect Agents using messages

Module 3
Reactive Extensions

Events in F#


Immutable
Implement Iobservable (Event-Stream)
First-Class
Composable (Combinators)

Event Combinators

The biggest benefit of using higher-order functions for events is that we can express the flow in a Declarative way

- **What** to do with received data using event **combinators**
- Use functions for working with *event values*

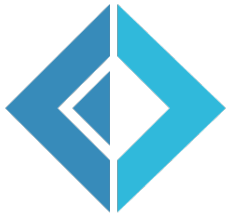
Seq Module

```
let myData = seq {1..10}

myData
    |> Seq.map(fun e -> e.ToString())
    |> Seq.filter(fun e -> e <> "")
    |> Seq.choose(fun e -> Some e)
```

Observalbe Module

```
let myEvent = Event<int>()

myEvent
    |> Event.map(fun e -> e.ToString())
    |> Event.filter(fun e -> e <> "")
    |> Event.choose(fun e -> Some e)
```

Event map & filter & add



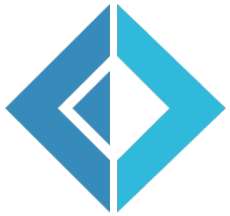
- Filter and trigger events in specific circumstances with `Event.filter`
- Create a new event that carries a different type of value with `Event.map`

```
Event.map    : ('T -> 'R)   -> IEvent<'T> -> IEvent<'R>
Event.filter : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
```

- Register event with `Event.add`

```
Event.add : ('T -> unit) -> IEvent<'Del,'T> -> unit
```

Event merge & scan



- Merging events with `Event.merge`
  - Triggered whenever first or second event occurs
  - Note that the carried values must have same type

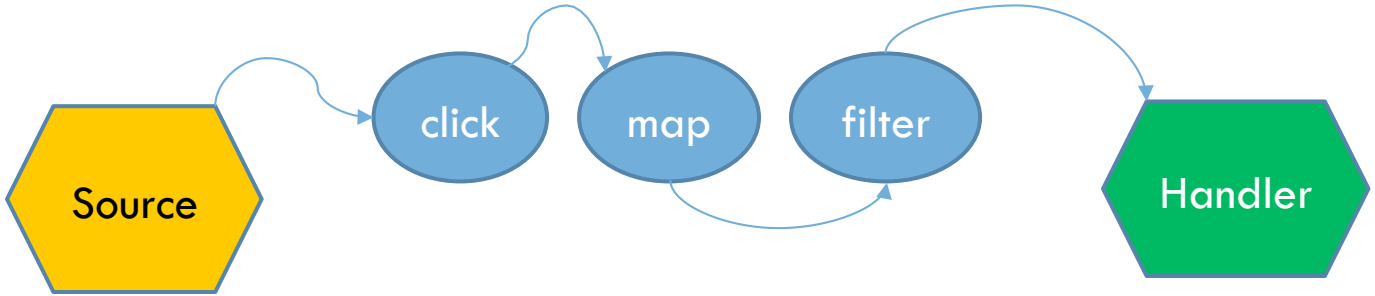
```
IEvent<'T> -> IEvent<'T> -> IEvent<'T>
```

- Creating stateful events with `Event.scan`
  - State is recalculated each time event occurs
  - Triggered with new state after recalculation

```
('St -> 'T -> 'St) -> 'St -> IEvent<'T> -> IEvent<'St>
```

Event are Observable… *almost*

- Memory leak ☹
  - IEvent does not support *removing* event handlers *(RemoveHandler on the resulting event, it leaves some handlers attached… leak!)*
  - Observable is able to remove handlers – **Idisposable`**

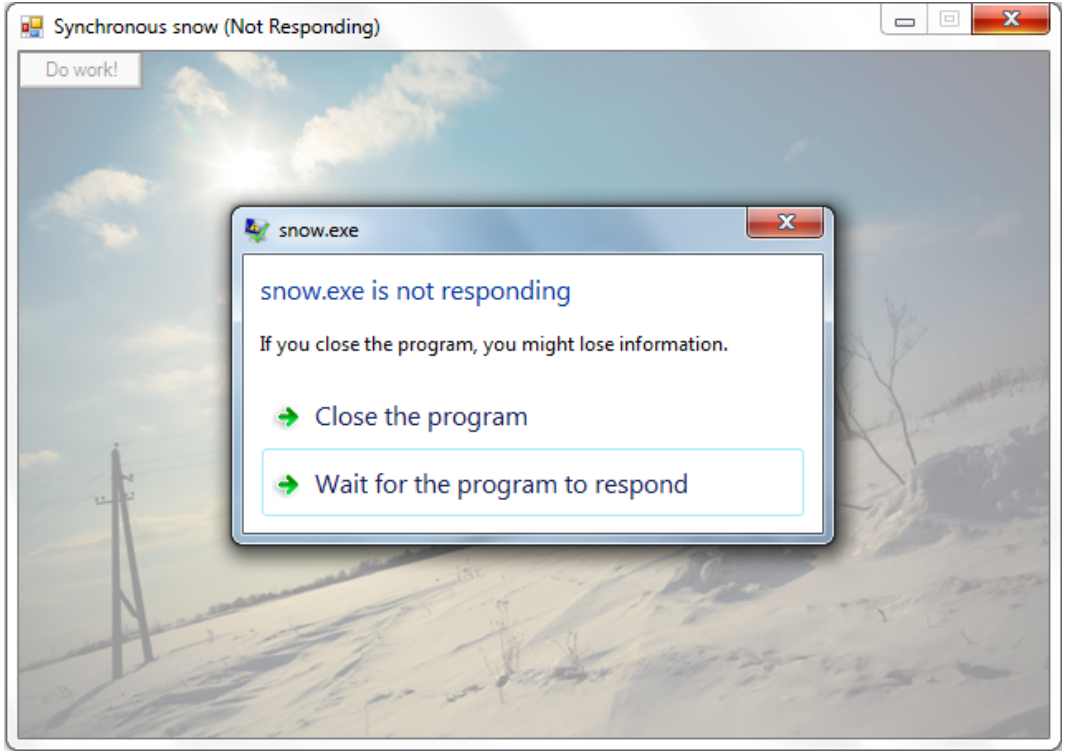


Observable in F#

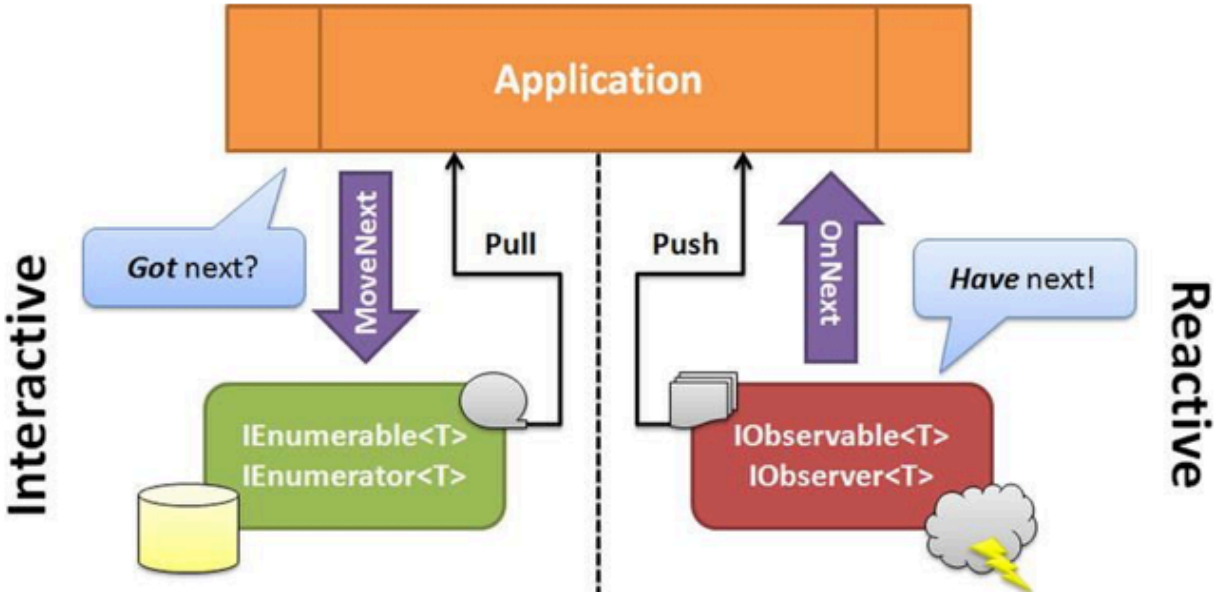
- Event<'T> interface inherits from IObservable<'T>
  - We can use the same standard F# Events functions for working with Observable

```
Observable.filter : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
Observable.map    : ('T -> 'R)   -> IObservable<'T> -> IObservable<'R>
Observable.add    : ('T -> unit)   -> IObservable<'T> -> unit
Observable.merge  : IObservable<'T> -> IObservable<'T> -> IObservable<'T>


      Observable.subscribe  : IObservable<'T> -> IDisposable
```

What are Reactive Extensions



Pull vs Push

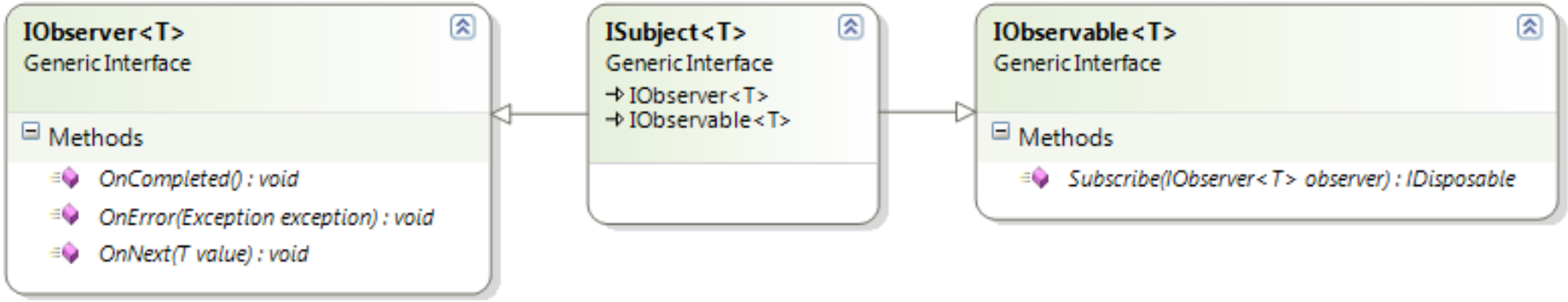

Application
Interactive
Got next?
MoveNext
Pull
IEnumerable<T>
IEnumerator<T>
Push
OnNext
Have next!
IObservable<T>
IObserver<T>
Reactive

IObserver & IObservable

```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

type IObservable<'a> = interface
    abstract Subsrcribe : IObserver<'a> -> IDisposable
end
```

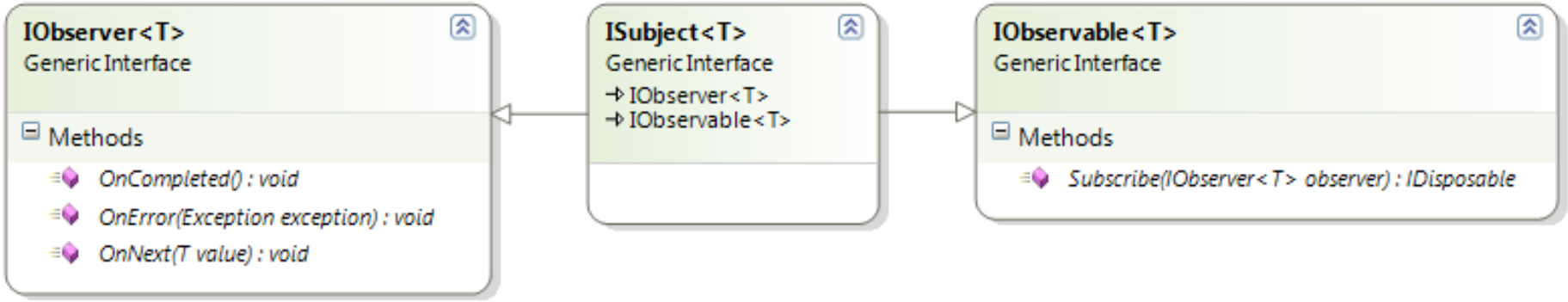
What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



What are Reactive Extensions

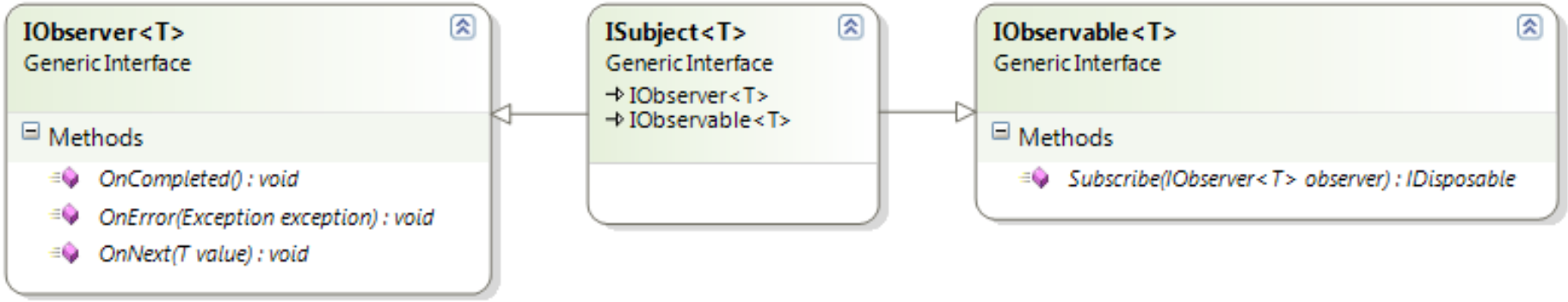
Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



publisher

What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators

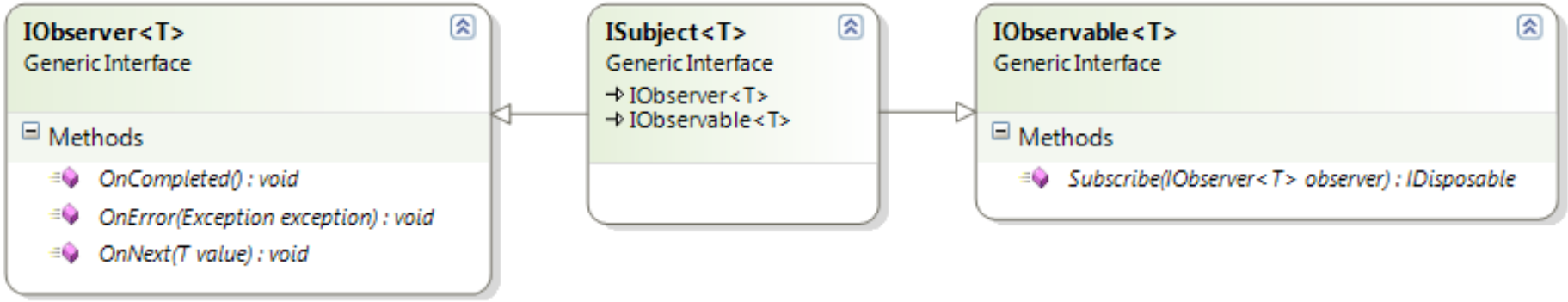


publisher

subscriber

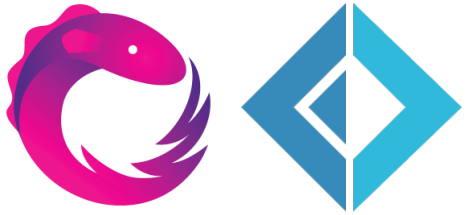
What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



publisher *both* *subscriber*

Subject<'a>.. as a bus



```
// Since RX is all about sequences of events/messages
// it does fit very well together with any sort of message bus or event broker.
type RxBusCommand() =
    let subject = new Subject<RxCommand>()

    member this.AsObservable() =
       subject.AsObservable()

    member this.Send(item:RxCommand) =
        subject.OnNext(item)


let busud = RxBusCommand()

busud.AsObservable().Do(fun t ->
                        match t with
                        | BuyTickets(name, quantity) -> printfn "I am buying %d tickets for %s" quantity name
                        | OrderDrink(drink) -> printfn "I am getting some %s to drink" drink).Subscribe() |> ignore

// The nice thing about this is that you get automatic Linq support since it is built into RX.
// So you can add message handlers that filters or transform messages.
busud.Send(BuyTickets("Opera", 2))
busud.Send(OrderDrink("Coke"))
```

```
type RxCommand =
    | BuyTickets of string * int
    | OrderDrink of string
```

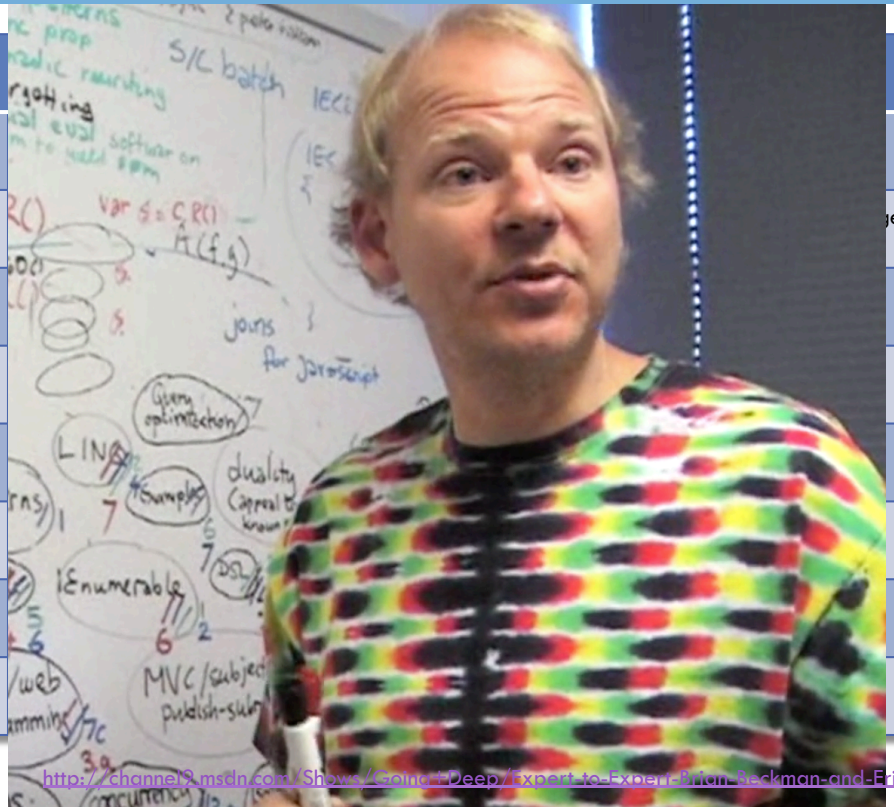
IObserver & IObservable

```
let observable = { new IObservable<string> with
                        member x.Subscribe(observer:IObserver<string>) = {
                           new IDisposable with
                              member x.Dispose() = ()
                         }
                    }

let observer = { new IObserver<string> with
                        member x.OnNext(value) = ()
                        member x.OnCompleted() = ()
                        member x.OnError(exn) = ()
                  }
```

F# & .Net RX Api

F# Observable
add
choose
filter
merge
pairwise
partition
scan
subscribe

.NET RX Api	
All	Amb
Combine	Count
Finally	First
Last	Latest
Multicast	Range
Switch	Select
Take	TakeLast
Window	When



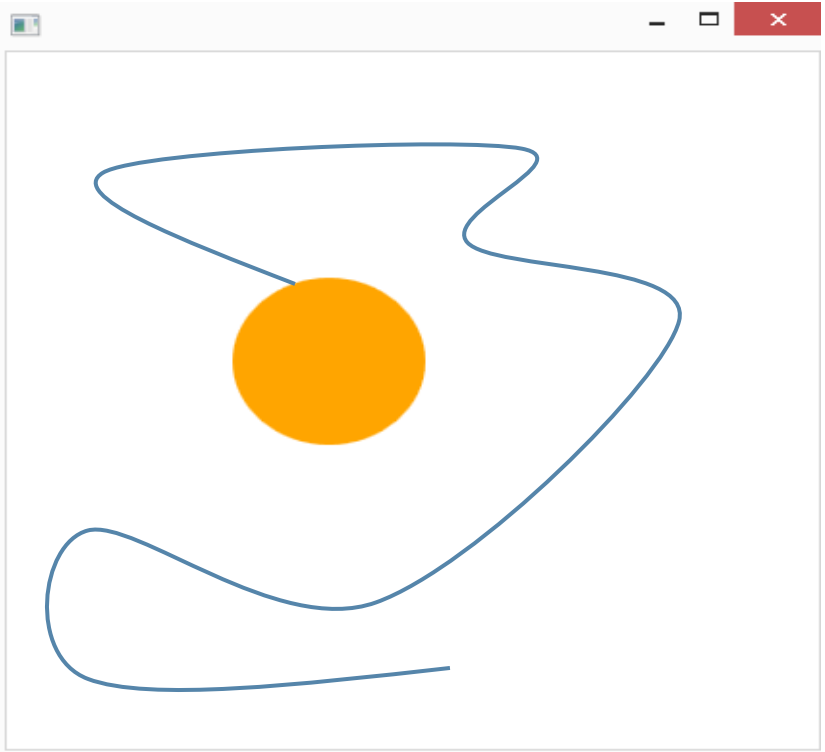
	Catch	Chunhify
ged	ElemntAt	ElementAtOrDefault
	If	IgnoreElement
	MinBy	MostRecent
	Never	Sum
	SkipUntill	Then
	While	When

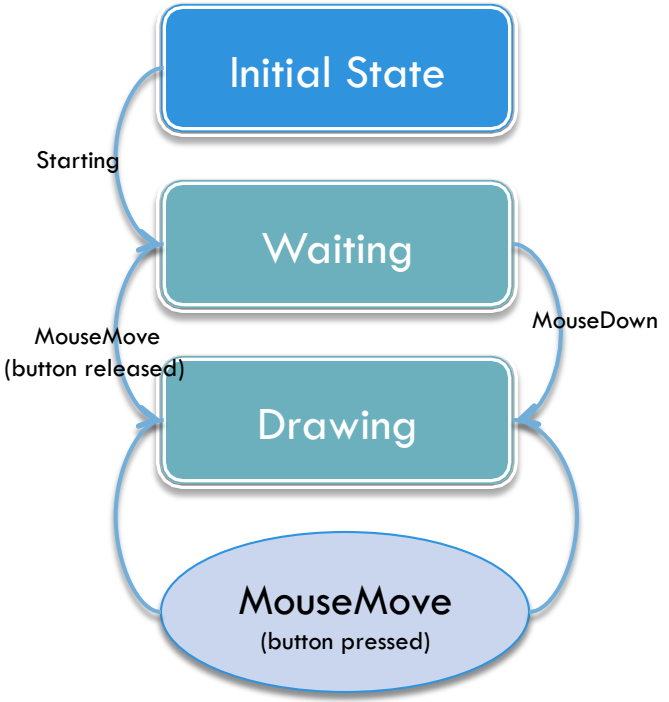
http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx

Workshop 3
Reactive Extensions

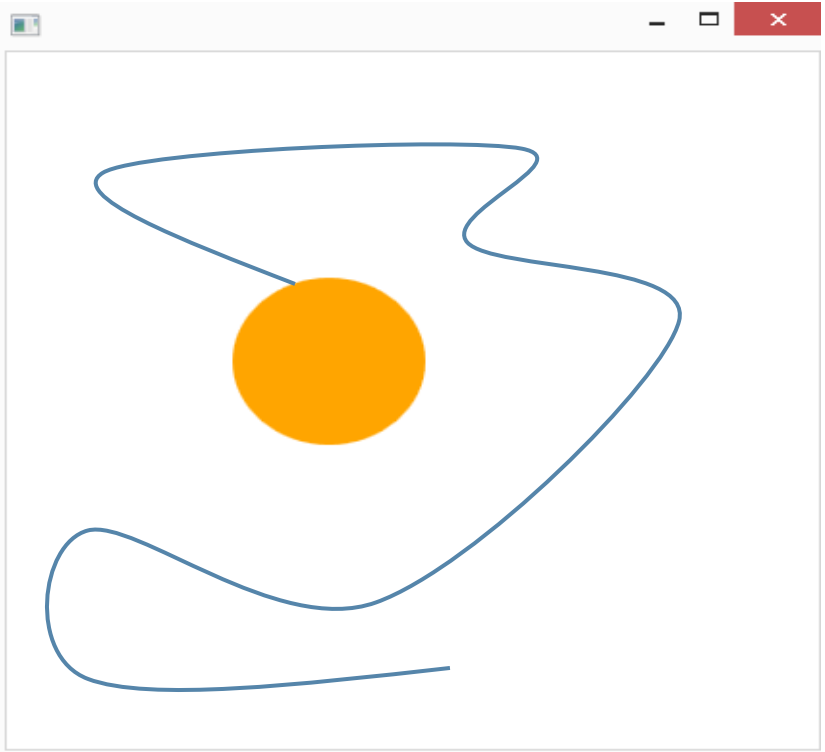
Estimated Time ~25 mins

Reactive Ball with undo





Reactive Ball with undo



Tasks

1. Add drawing line functionality when mouse is pressed
2. While drawing the line collect coordinates
3. Create undo (memento pattern) functionality.
    1. When the mouse is release, the Ball should replay backward the path following the line (slowly for animation)

Summary

Today developers must embrace concurrent programming

Functional Programming helps to go Reactive

F# really **shines** in the area of distributed computing

Get motivation to practice and master F# concurrent programing model

Q & A ?

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

How to reach me

github.com/rikace/GothamFS

meetup.com/DC-fsharp

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!