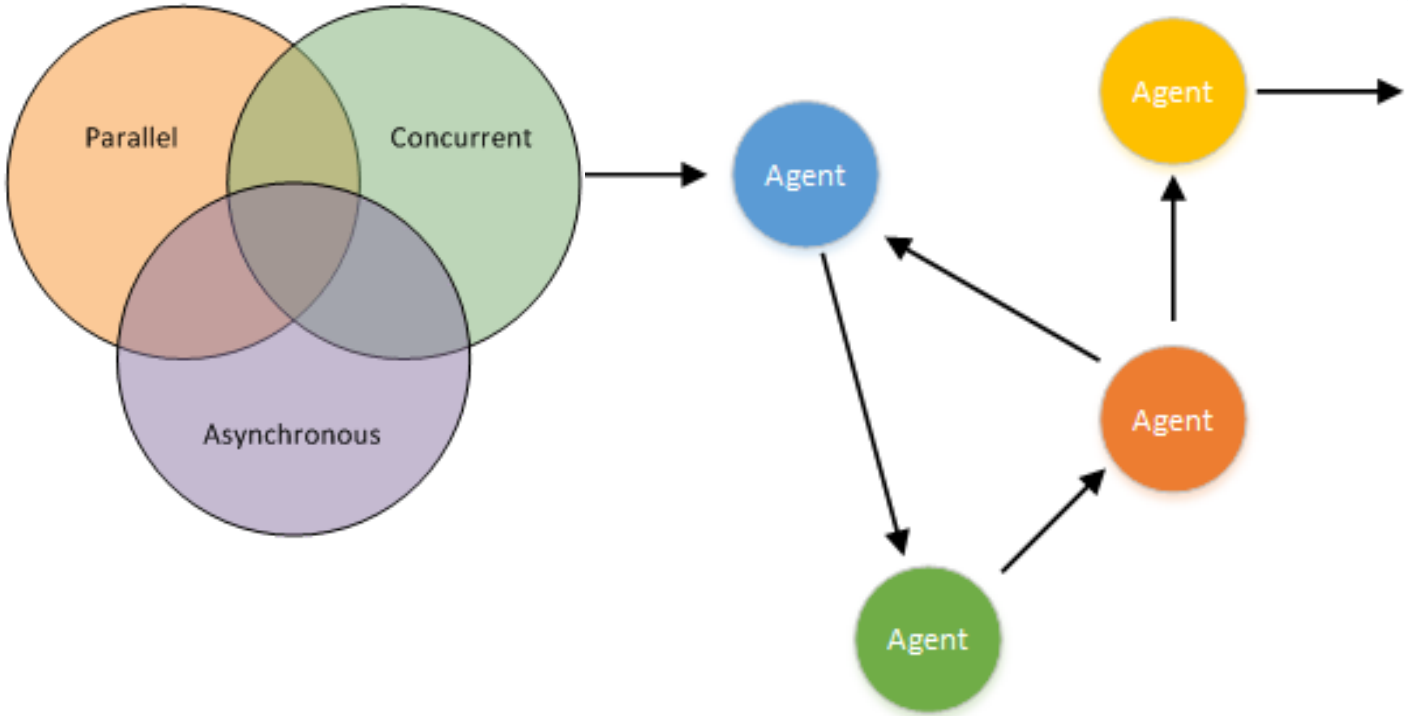
Programming Concepts and Languages

Spring 2024

Learning Objectives

- ✓ explain F# Concurrent Programming
  - ✓ implement concurrent programs using the agent/actor model approach to concurrency

F# Concurrent Programming



Agent/Actor Programming

- F# supports a variation of the Actor programming model through the in-memory implementation of lightweight asynchronous agents.

- similar to Akka for Scala and agents in Erlang, but unlike the Erlang ones, they do *not* work across process boundaries, only in the same process.

Actor Model

- a model of message-based concurrent computation which treats “actors" (aka “agents") as the universal primitives
- when a message is received, an actor can make local decisions, create more actors send more messages (change state) determine how to respond to the next message received
- no assumed sequence to the above actions and they could be carried out in parallel
- Theoretically, it could take an unbounded time for a message sent to be received

MailboxProcessor Actor/Agent

- The agent encapsulates an “inbox" message queue that supports multiple-writers and a single reader (the agent itself)
- Writers can send one-way messages to the agent by using the Post method and its variations
- Writers can send two-way messages (i.e. messages which request replies) to the agent by using variations of the PostAndReply method
- Agents can receive messages using the Receive method and its variations
- … and more

MailboxProcessor I

- All communication with agents is handled using message passing
- Messages are typically discriminated unions

- The message types:

```
type Msg1 = ... // one-way to-agent messages

type Reply = ... // one-way from-agent messages

type Msg2 = ... AsyncReplyChannel<Reply>... // two-ways
send/reply messages

 type Msg = j Msg1 j Msg2 // all to-agent messages
```

MailboxProcessor - Post

- Ideally, all agent's interactions with the external world, including its “result", are performed by send/reply messages (unless the agent has side-effects)
- To post a one-way message, (msg1:Msg1), to actor a, via a technically synchronous call, but in fact the start of an asynchronous process

```
agent.Post message
```

- To receive a message (msg:Msg), via an asynchronous awaited call, with optional timeout

```
let! message = inbox.Receive() // Async<Message>
```

MailboxProcessor – Post II

- To post a two-way message, (msg2:Msg2), to actor a, and wait for the reply (rep :Reply)
  - blocking call, with optional timeout

```
// Reply
let rep = agent.PostAndReply(fun repChan -> message2)
```

- asynchronous awaited call, with optional timeout

```
// Async<Reply>
let! rep = agent.PostAndAsyncReply(fun repChan -> message2
```

- construct message2, by including the reply channel repChan: AsyncReplyChannel<Reply>, *given by the runtime system as argument to your function*

MailboxProcessor - Start

- The static factory function ***Start*** is a typical way to create and start an agent:

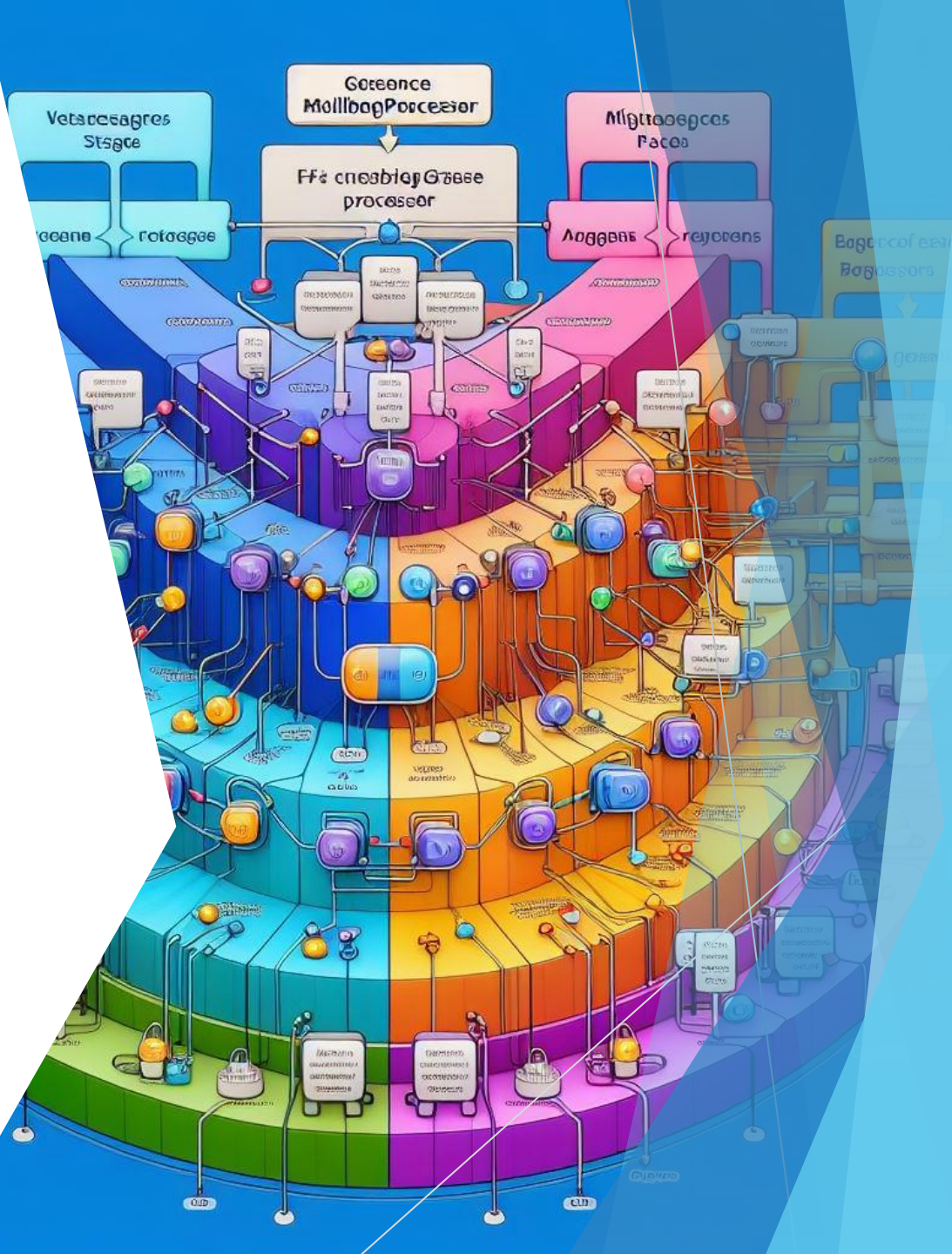
```
let agent = MailboxProcessor.Start
                (fun inbox -> async {...})
```

- Your task is to define a ***Start*** parameter: which is a "constructor" function `(MailboxProcessor<Msg> -> Async<unit>)`

- The "inbox" parameter represents the input message queue
- You do NOT create the "inbox" argument: *the runtime invokes your "constructor" function with the proper argument*
- The created agent has the same type as its "inbox", and its actual body has type Async<unit>

Examples

- Mailboxprocessor



MailboxProcessor - Example 1

- Implementing a printer agent

```
// MailboxProcessor
// A simple print agent
let printAgent =
    MailboxProcessor.Start (fun inbox ->
        // a function to process the message in the
inbox
        let rec msgLoop = async {
            // read a message
            let! msg = inbox.Receive()
            // process the message
            printfn "\nThe message is: %s" msg
            // loop to the top
            return! msgLoop
        }
        // start the loop
        msgLoop)
```

MailboxProcessor - Example 2

- Implementing a message processing actor

```
// A case converter
let caseConverterAgent =
    MailboxProcessor.Start (fun inbox ->
        // a function to process the message in the inbox
        let rec processMessage state = async {
            // read a message
            let! msg = inbox.Receive()
            // process the message
            printfn "\nReceived:  %A" msg
            let newState = "["  + msg + "]"
            printfn "Processed: %s" (newState.ToUpper())
            // loop to the top
            return! processMessage newState
        }
        // start the loop
        processMessage "initialState")

let data = ["apple";"banana";"carrot";"durian";"elgray";"fruit"]
data |> List.map caseConverterAgent.Post
(*
> caseConverterAgent.Post "abracadabra";;
Received:  val "abracadabra"
Processed: [ABRACADABRA]
*)
```

Next week Presentation

Group	Student name	Group name
1	Catalin Filip Marius Babin Roberts Zustars	
2	Karolis Sadeckas Arturs Silins	
3	Himal Sharma Sachin Baral	Duo Levelling
4	Anders Nørgaard Blumensaat Mikkel Rumle Bøie Winther Sebastian Peter Ørndrup	
5	Ion Canariov Juan Vizcaino Martin Lyuboslav Lyubomirov Kotsev	
6	Alfonso Pedro Ridao Mihai Avram	
10	Adam Arasimowicz Esben Vensel Fogh	
11	Emil Rumenov Vasilev Martynas Vycas	
12	Adrian-Cristian Militaru Cristian-Marian Radu	
	Alexandru Cotruta	