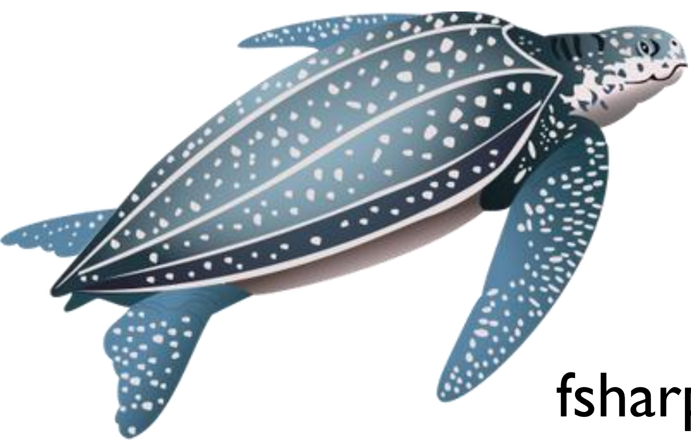
I'll be using F# code examples, but the concepts will work in most programming languages.

Thirteen ways of looking at a turtle



@ScottWlaschin

fsharpforfunandprofit.com/turtle

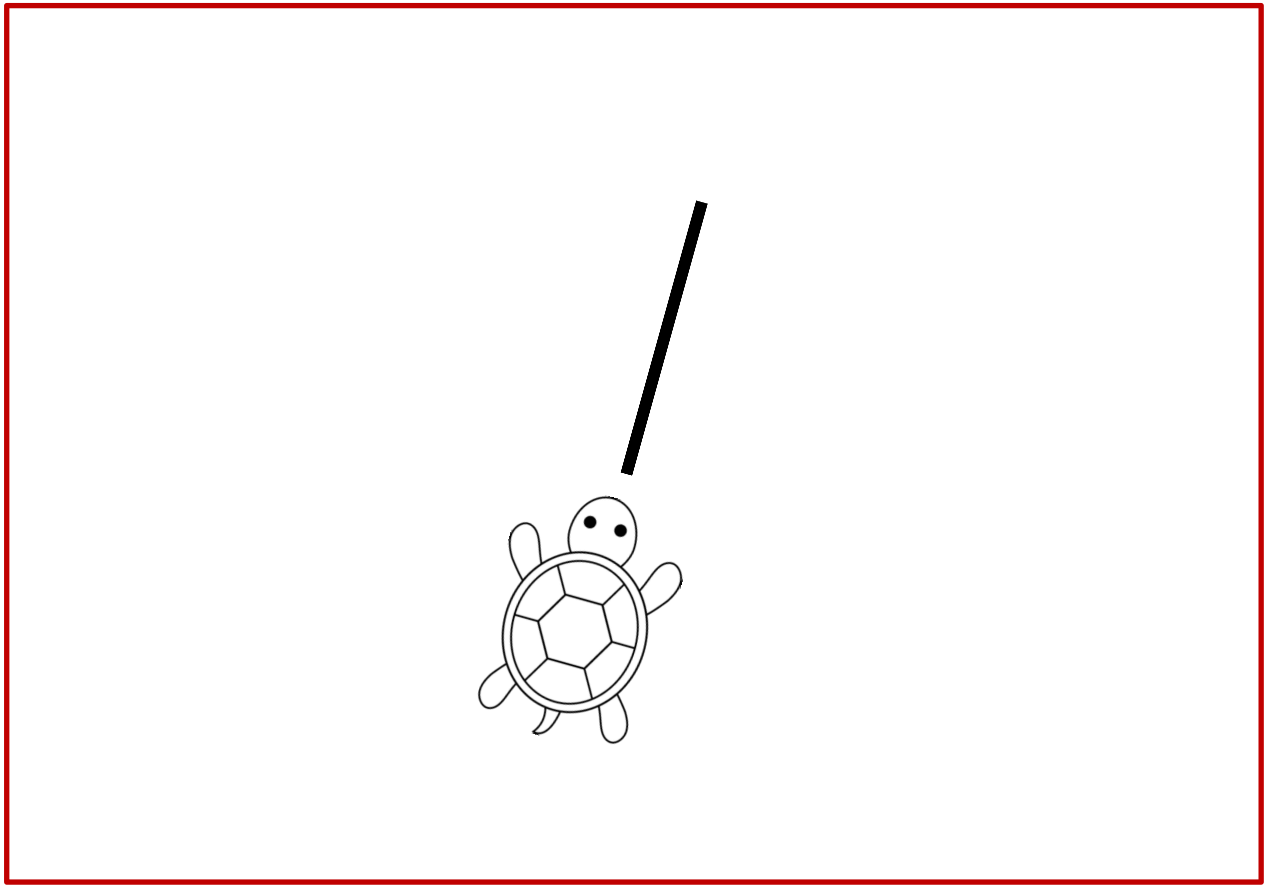
A taste of many different approaches

A taste of many different approaches

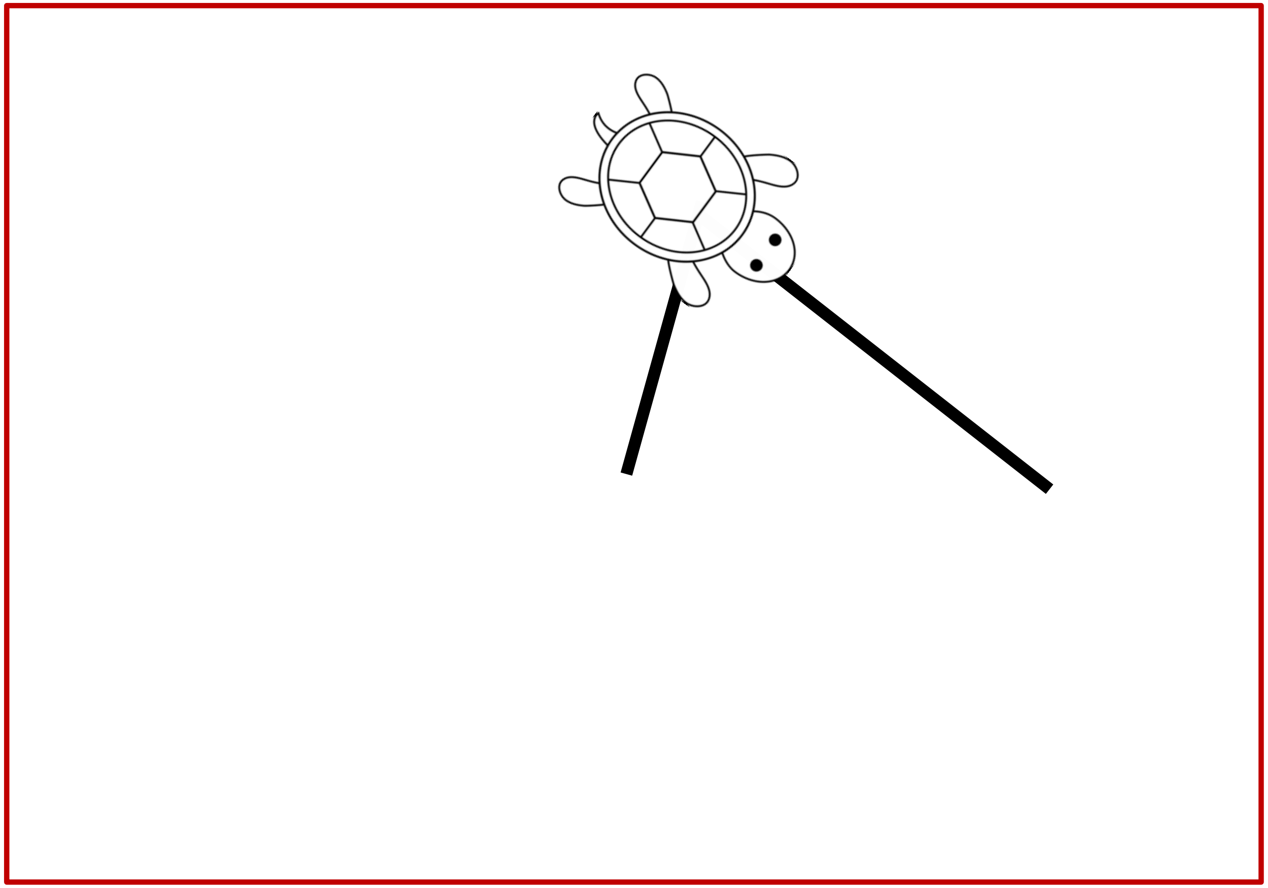
This is a crazy experiment:
~4 mins per topic!

See also fsharpforfunandprofit.com/fppatterns

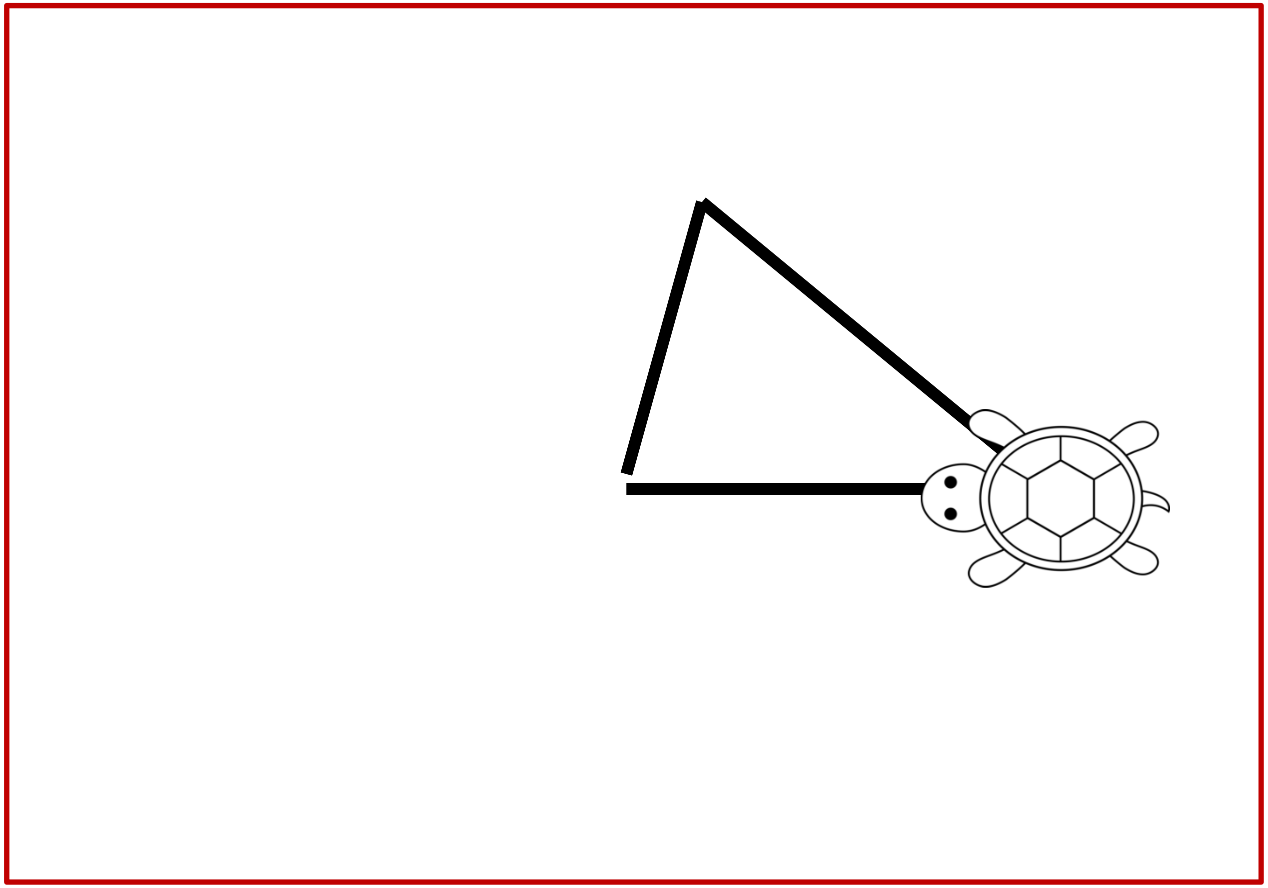
Turtle graphics in action



Turtle graphics in action



Turtle graphics in action



Turtle API

API	Description
`Move aDistance`	Move some distance in the current direction.
`Turn anAngle`	Turn a certain number of degrees clockwise or anticlockwise.
`PenUp` `PenDown`	Put the pen down or up. When the pen is down, moving the turtle draws a line.

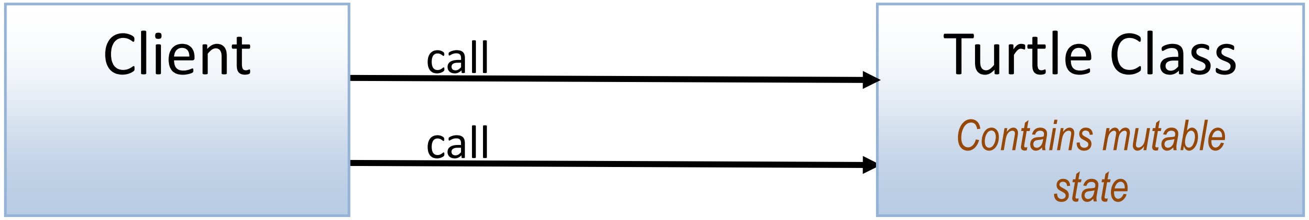
All of the following implementations will be based on this interface or some variant of it.

1. Object Oriented Turtle

A "Tootle"

Data and behavior are combined

Overview



```
type Turtle() =

    let mutable currentPosition = initialPosition
    let mutable currentAngle = 0.0<Degrees>
    let mutable currentPenState = initialPenState
```

"mutable" keyword needed in F#

Units of measure used to document API

```
member this.Move(distance) =
    Logger.info (sprintf "Move %0.1f" distance)
    let startPos = currentPosition
    // calculate new position
    let endPos = calcNewPosition distance currentAngle startPos
    // draw line if needed
    if currentPenState = Down then
        Canvas.draw startPos endPos
    // update the state
    currentPosition <- endPos
```

Assignment

```
member this.Turn(angleToTurn) =
    Logger.info (sprintf "Turn %0.1f" angleToTurn)

    // calculate new angle
    let newAngle =
        (currentAngle + angleToTurn) % 360.0<Degrees>

    // update the state
    currentAngle <- newAngle
```

```
member this.PenUp() =
    Logger.info "Pen up"
    currentPenState <- Up

member this.PenDown() =
    Logger.info "Pen down"
    currentPenState <- Down
```

Usage example

```
let drawTriangle() =
    let distance = 50.0
    let turtle = Turtle()
    turtle.Move distance
    turtle.Turn 120.0<Degrees>
    turtle.Move distance
    turtle.Turn 120.0<Degrees>
    turtle.Move distance
    turtle.Turn 120.0<Degrees>
    // back home at (0,0) with angle 0
```

OO Turtle demo

Advantages and disadvantages

- Advantages
  - Familiar
- Disadvantages
  - Stateful/Black box
  - Can't easily compose
  - Hard-coded dependencies (for now)

2. Abstract Data Turtle

Data is separated from behavior

Data

```
type TurtleState = private {
    mutable position : Position
    mutable angle : float<Degrees>
    mutable penState : PenState
}
```

Only turtle functions can access it

Behavior

```
module Turtle =
    let move distance state =  ...
    let turn angleToTurn state = ...
    let penUp state = ...
    let penDown log state =
```

State passed in explicitly

Usage example

```
let drawTriangle() =
    let distance = 50.0
    let turtle = Turtle.create()
    Turtle.move distance turtle
    Turtle.turn 120.0<Degrees> turtle
    Turtle.move distance turtle
    Turtle.turn 120.0<Degrees> turtle
    Turtle.move distance turtle
    Turtle.turn 120.0<Degrees> turtle
```

State passed in explicitly

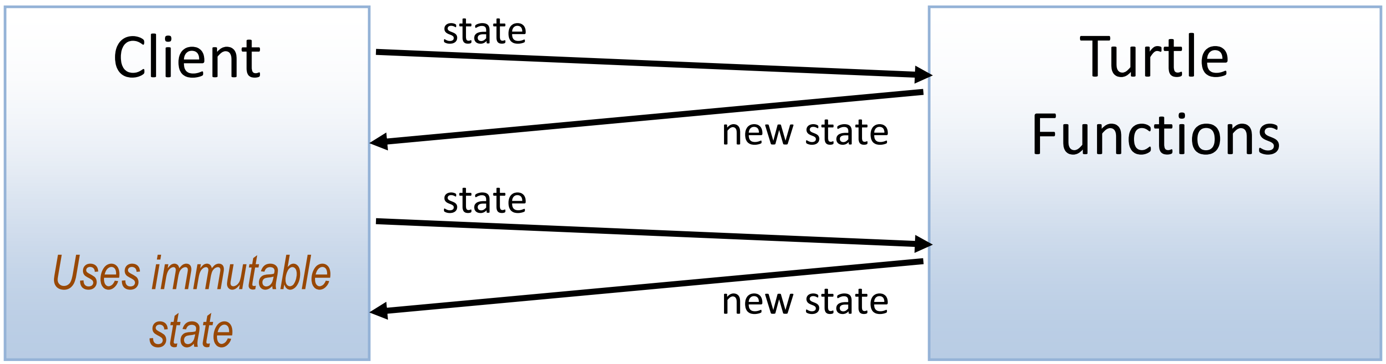
Advantages and disadvantages

- Advantages
  - Simple
  - Forces composition over inheritance!
- Disadvantages
  - As with OO: stateful, etc

3. Functional Turtle

Data is immutable

Overview



Client
Uses immutable state
state
new state
state
new state
Turtle Functions

Data

```
type TurtleState = {
    position : Position
    angle : float<Degrees>
    penState : PenState
}
```

Public, immutable

Behavior

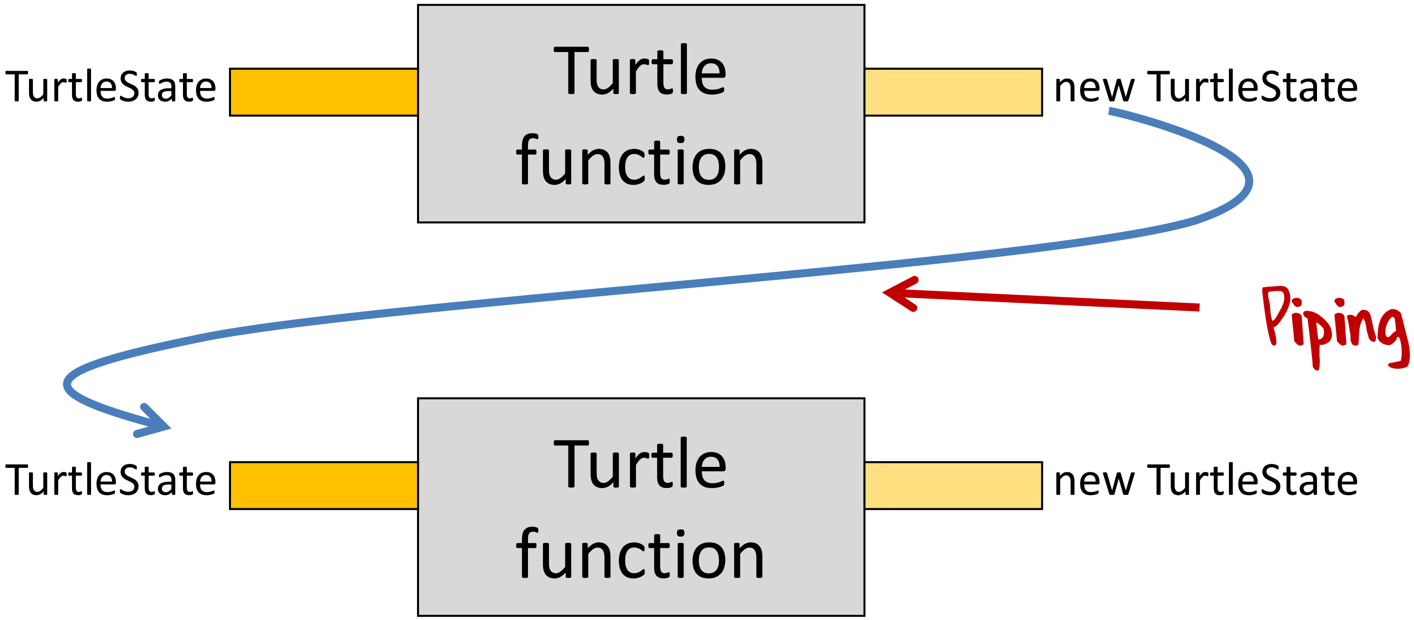
```
module Turtle =
    let move distance state =  ... // return new state
    let turn angleToTurn state = ... // return new state
    let penUp state = ... // return new state
    let penDown log state = // return new state
```

State passed in explicitly AND returned

Usage example

```
let drawTriangle() =
    let s0 = Turtle.initialTurtleState
    let s1 = Turtle.move 50.0 s0
    let s2 = Turtle.turn 120.0<Degrees> s1
    let s3 = Turtle.move 50.0 s2
    ...
```

Passing state around is annoying and ugly!


TurtleState
Turtle function
new TurtleState
Piping
TurtleState
Turtle function
new TurtleState

Usage example with pipes

```
let drawTriangle() =
    Turtle.initialTurtleState
    |> Turtle.move 50.0
    |> Turtle.turn 120.0<Degrees>
    |> Turtle.move 50.0
    ...
```

|> is pipe operator

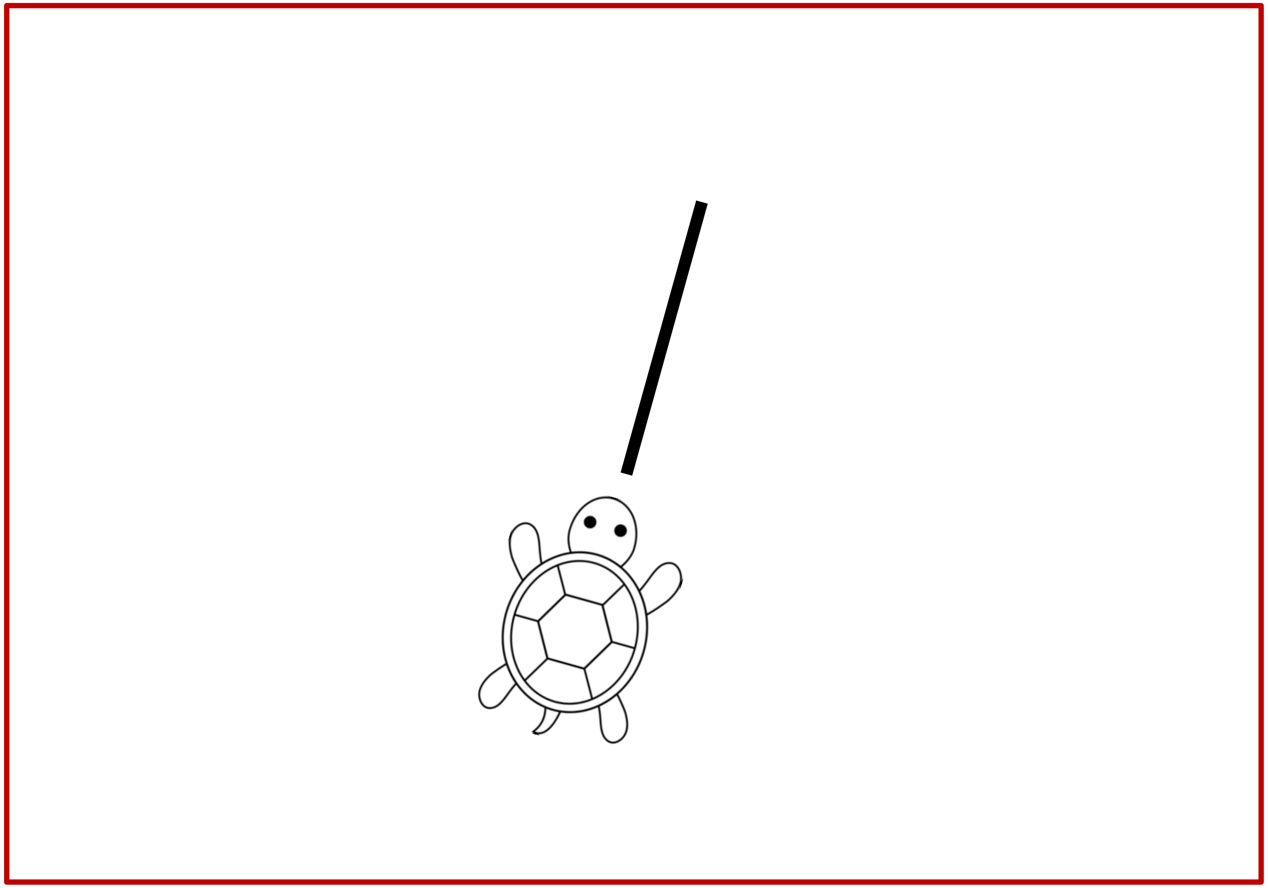
Advantages and disadvantages

- Advantages
  - Immutability: Easy to reason about
  - Stateless: Easy to test
  - Functions are composable
- Disadvantages
  - Client has to keep track of the state ☹
  - Hard-coded dependencies (for now)

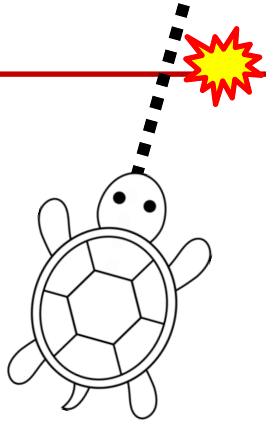
4. State monad

Threading state behind the scenes

Turtle Canvas



Turtle Canvas



Change implementation so that Move returns a pair:
* New state, and
* Actual distance moved

Usage example

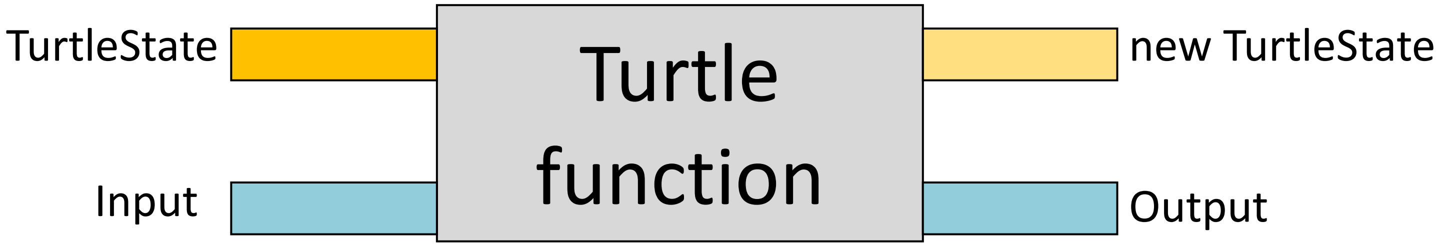
The returned pair

```
let s0 = Turtle.initialTurtleState
let (actualDistA,s1) = Turtle.move 80.0 s0
if actualDistA < 80.0 then
    printfn "first move failed -- turning"
    let s2 = Turtle.turn 120.0<Degrees> s1
    let (actualDistB,s3) = Turtle.move 80.0 s2
    ...
else
    printfn "first move succeeded"
    let (actualDistC,s2) = Turtle.move 80.0 s1
    ...
```

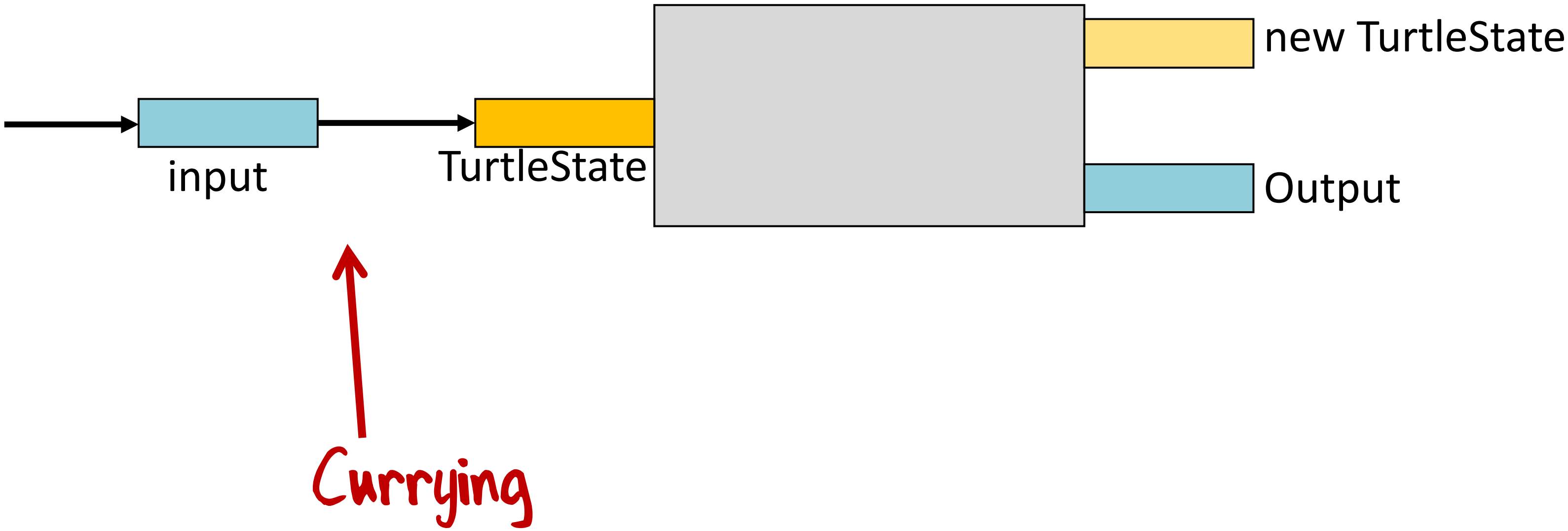
Yuck!

How can we keep track of the state?

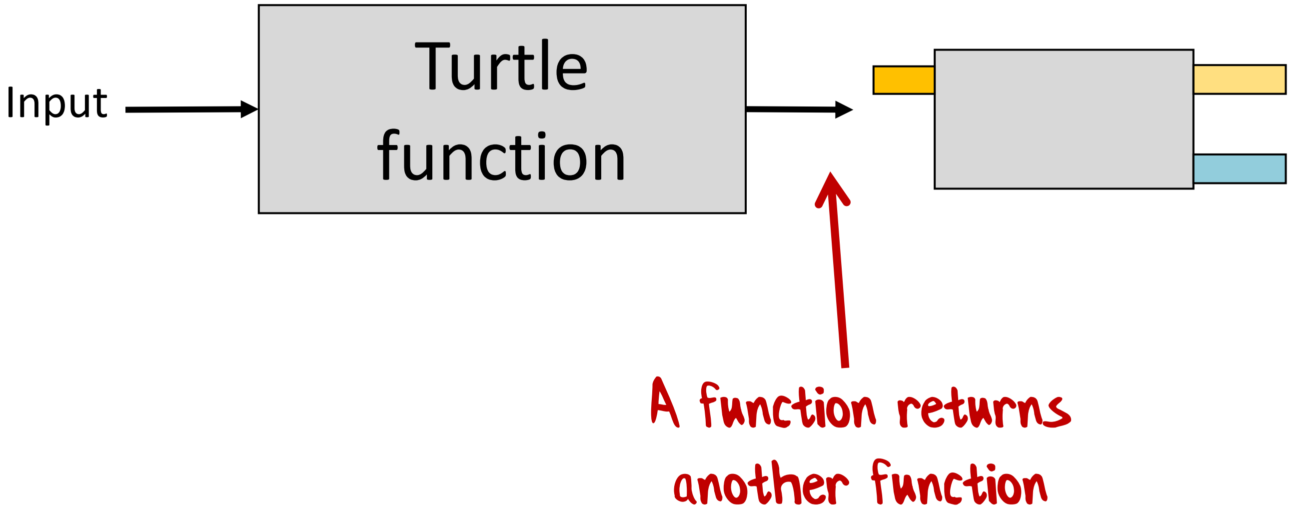
Transforming the function #1



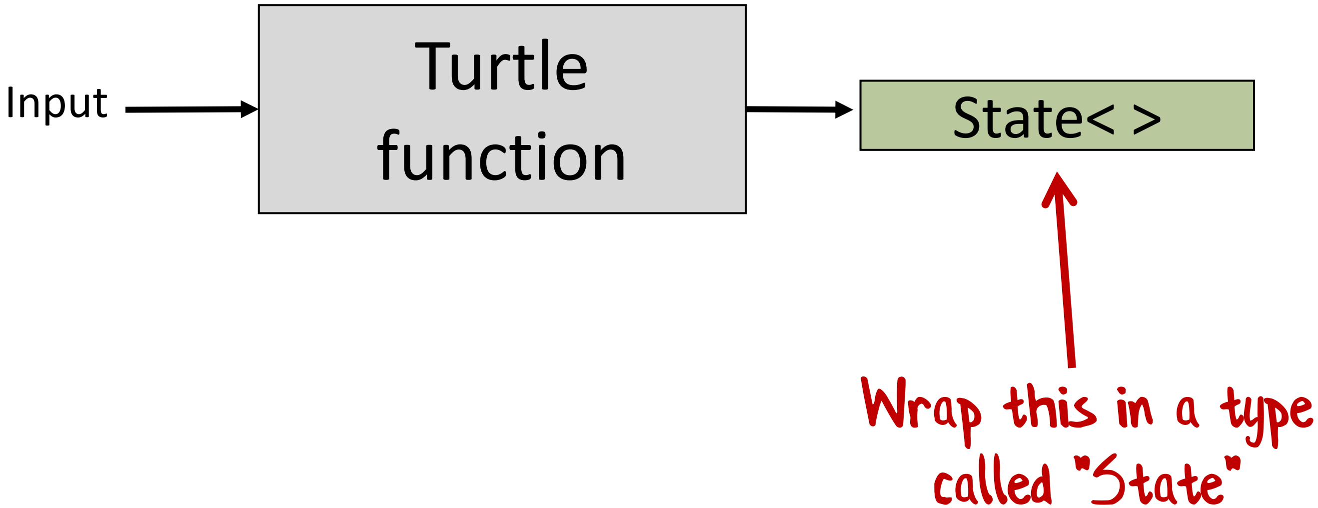
Transforming the function #2



Transforming the function #3



Transforming the function #4



For more on how this works, see fsharpforfunandprofit.com/monadster

Usage example

```
let stateExpression = state {              // "state" expression
    let! distA = move 80.0
    if distA < 80.0 then
        printfn "first move failed -- turning"
        do! turn 120.0<Degrees>
        let! distB = move 80.0
        ...
    else
        printfn "first move succeeded"
        let! distB = move 80.0
        ...
}
```

State is threaded through behind the scenes

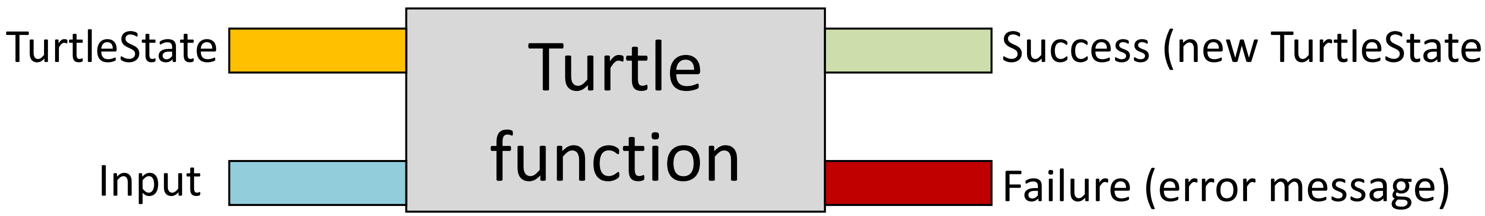
Haskell has "do" notation. Scala has "for" comprehensions

Advantages and disadvantages

- Advantages
  - Looks imperative, but preserves immutability.
  - Functions still composable
- Disadvantages
  - Harder to implement and use

5. Error handling

How to return errors?




```
type Result<'successInfo,'errorInfo> =
    | Success of 'successInfo
    | Failure of 'errorInfo
```

Choice (aka Sum, aka Discriminated Union) type

Implementation using Result

```
let move distanceRequested state =
    // calculate new position
    // draw line if needed
    if actualDistanceMoved <> distanceRequested then
        Failure "Moved out of bounds"
    else
        Success {state with position = endPosition}
```

Two different choices for return value
(not easy in OO)

Usage example

```
let s0 = Turtle.initialTurtleState
let result1  = s0 |> Turtle.move 80.0
match result1  with
| Success s1  ->
    let result2 = s1 |> Turtle.move 80.0
    match result2 with
    | Success s2 ->
        printfn "second move succeeded"
        ...
    | Failure msg ->
        printfn "second move failed: %s" msg
        ...
| Failure msg ->
    printfn "first move failed -- %s" msg
```

Again Yuck!

Usage example

```
let finalResult = result {               // result expression
    let s0 = Turtle.initialTurtleState
    let! s1 = s0 |> Turtle.move 80.0
    printfn "first move succeeded"
    let! s2 = s1 |> Turtle.move 30.0
    printfn "second move succeeded"
    let! s3 = s2 |> Turtle.turn 120.0<Degrees>
    let! s4 = s3 |> Turtle.move 80.0
    printfn "third move succeeded"
    return ()
    }
```

Errors are managed behind the scenes

Combine "state" and "result" for even prettier code

Usage example

```
let finalResult = resultState {         // "result" and "state" combined
        do! Turtle.move 80.0
        printfn "first move succeeded"
        do! Turtle.move 30.0
        printfn "second move succeeded"
        do! Turtle.turn 120.0<Degrees>
        do! Turtle.move 80.0
        printfn "third move succeeded"
        return ()
        }
```

Both errors and state are now managed behind the scenes

Advantages and disadvantages

- Advantages
  - Errors are explicitly returned (no exceptions)
  - Looks like "happy path" code
- Disadvantages
  - Slightly harder to implement and use

5½. Async turtle

What if the Turtle calls were async?

What if the Turtle calls were async?

```
let s0 = Turtle.initialTurtleState
s0 |> Turtle. moveAsync 80.0 (fun s1 ->
    s1 |> Turtle.moveAsync 80.0 (fun s2 ->
        s2 |> Turtle.moveAsync 80.0 (fun s3 ->
        ...
        )
    )
)
```

Callbacks

Yuckety Yuck!

Async usage example

```
let finalResult = async {                    // "async" expression
    let s0 = Turtle.initialTurtleState
    let! s1  = s0 |> Turtle.moveAsync 80.0
    let! s2 = s1  |> Turtle. moveAsync 80.0
    let! s3 = s2 |> Turtle. moveAsync 80.0
     ...
    }
```

Callbacks are managed behind the scenes

Do you see a pattern?
(the m-word)

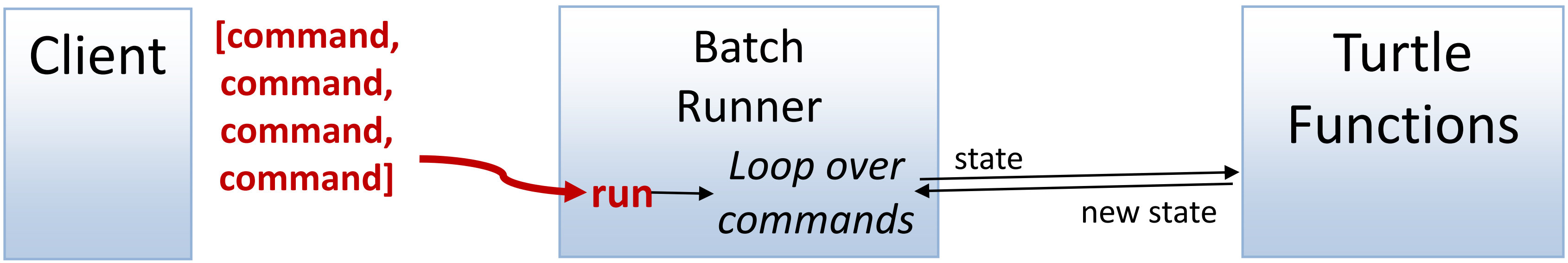
Review: Common patterns

- Composition
  - Chaining functions together
- Explicitness
  - Explicit state management (no mutation)
  - Explicit errors (no exceptions)
- Techniques to thread state/errors/callbacks behind the scenes (the m-word)

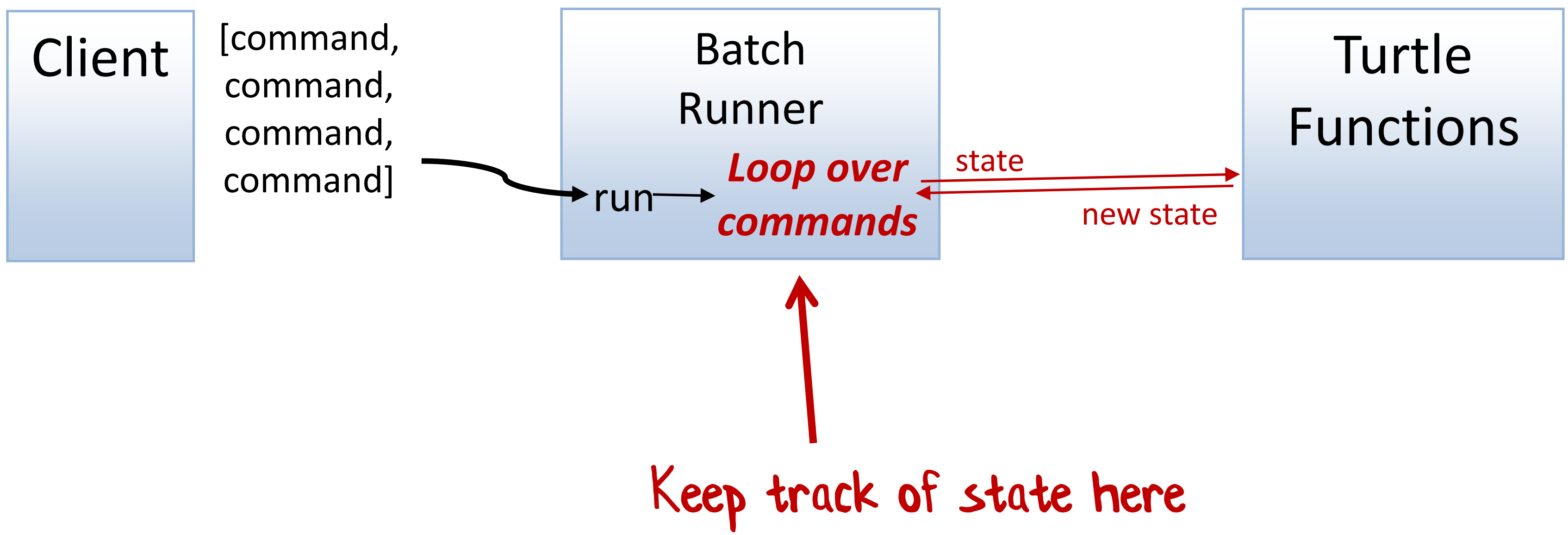
6. Batch commands

How can the caller avoid managing state?

Overview


Client
[command,
command,
command,
command]
Batch
Runner
run
Loop over
commands
state
new state
Turtle
Functions

Overview


Client
[command,
command,
command,
command]
Batch
Runner
run
Loop over
commands
state
new state
Turtle
Functions
Keep track of state here

How to convert a function into data?

```
// Turtle functions
let move distance = ...
let turn angle = ...
let penUp () = ...
let penDown () = ...
```

```
type TurtleCommand =
| Move of Distance
| Turn of Angle
| PenUp
| PenDown
```

Choice type

Usage example

```
// create the list of commands
let commands = [
       Move 100.0
       Turn 120.0<Degrees>
       Move 100.0
       Turn 120.0<Degrees>
       Move 100.0
       Turn 120.0<Degrees>
       ]
// run them
TurtleBatch.run commands
```

This is *data* not function calls

"*execute*" *implementation*

```
/// Apply a command to the turtle state and return the new state
let executeCommand state command =
    match command with
    | Move distance ->
      Turtle.move distance state
    | Turn angleToTurn ->
      Turtle.turn angleToTurn state
    | PenUp ->
      Turtle.penUp state
    | PenDown ->
      Turtle.penDown state
```

On-to-one correspondence

"*run*" *implementation*

```
/// Run list of commands in one go
let run aListOfCommands =
    let mutable state = Turtle.initialTurtleState
    for command in aListOfCommands do
        state <- executeCommand state command
    // return final state
    state
```

"*run*" *implementation*

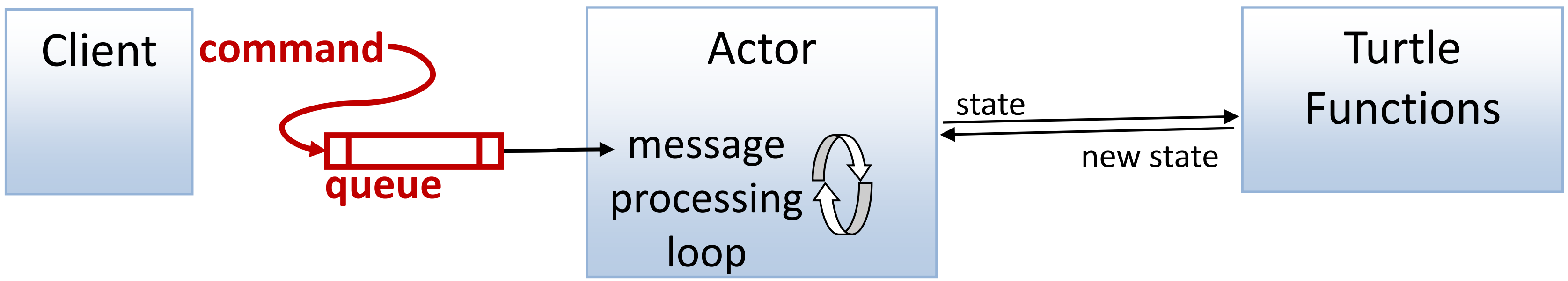
```
/// Run list of commands in one go
let run aListOfCommands =
    aListOfCommands
    |> List.fold executeCommand Turtle.initialTurtleState
```

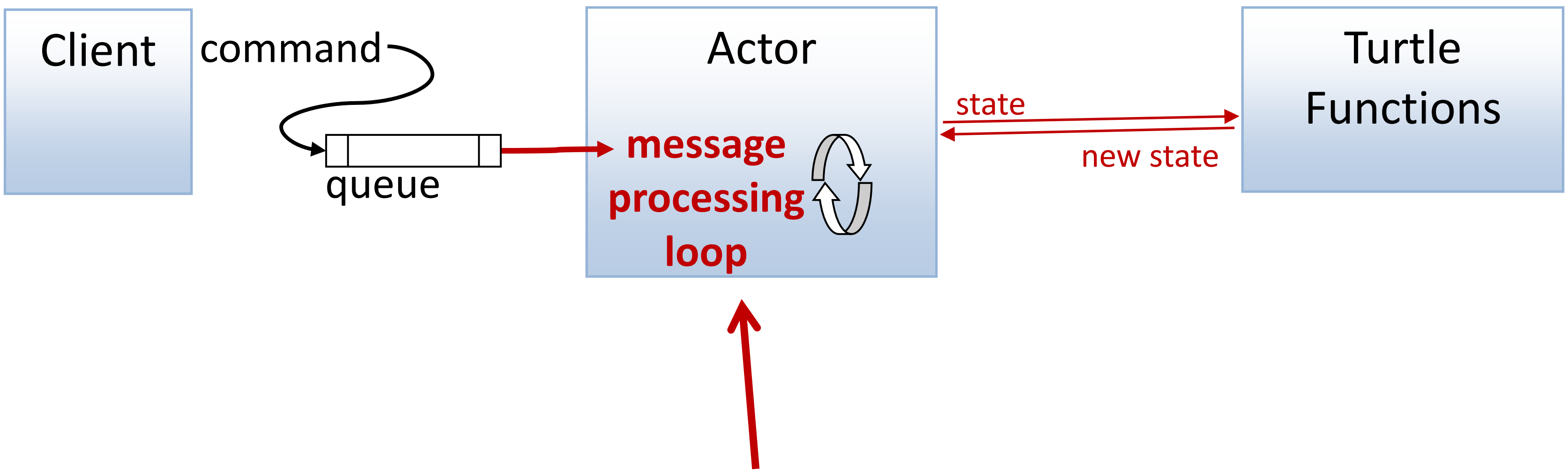
Use built-in collection functions
where possible

Advantages and disadvantages

- Advantages
  - Decoupled
  - Simpler than monads!
- Disadvantages
  - Batch oriented only
  - No control flow inside batch

7. Actor model


Client
command
queue
Actor
message
processing
loop
state
new state
Turtle
Functions



Keep track of state here

Pulling commands off a queue

```
let rec loop turtleState =
    let command = // read a command from the message queue
    let newState =
        match command with
        | Move distance ->
            Turtle.move distance turtleState
        // etc
    loop newState
```

Like batch implementation

Recurse with new state

Usage example

```
// post a list of commands
let turtleActor = new TurtleActor()
turtleActor.Post (Move 100.0)
turtleActor.Post (Turn 120.0<Degrees>)
turtleActor.Post (Move 100.0)
turtleActor.Post (Turn 120.0<Degrees>)
```

Again, this is *data*

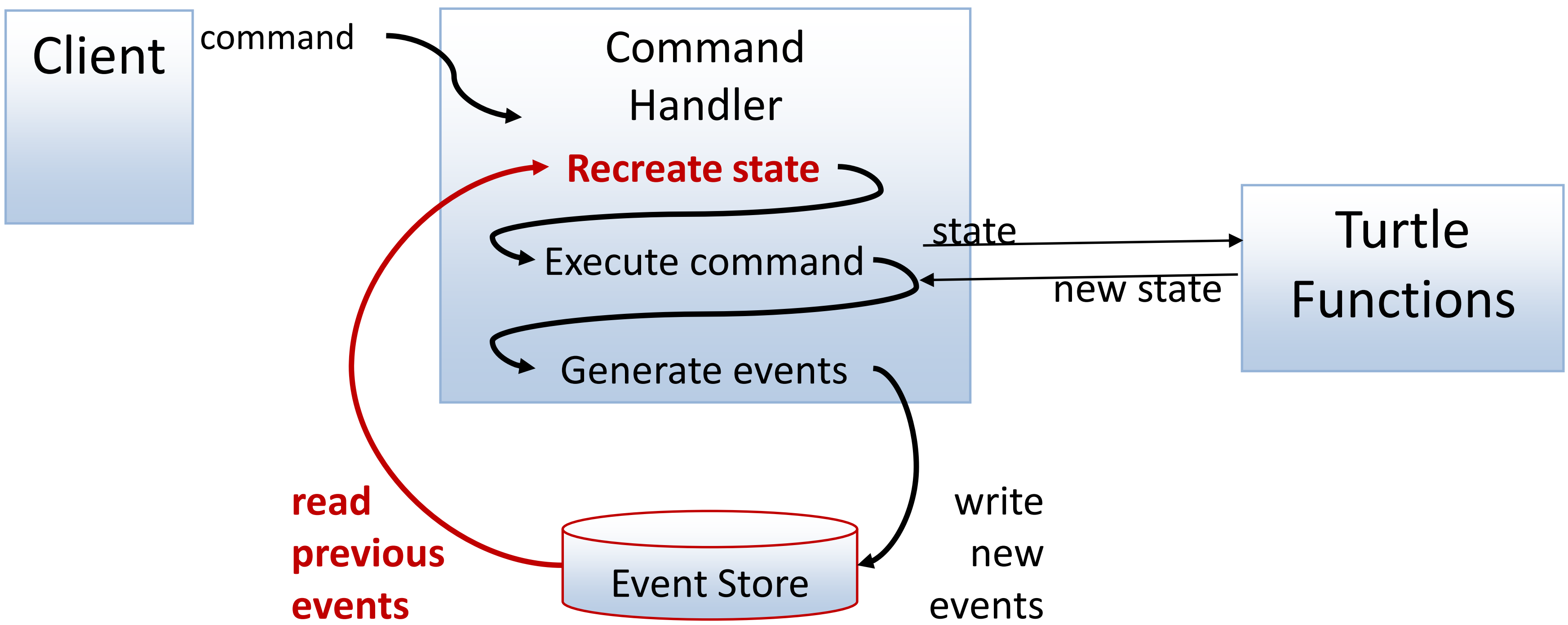
Advantages and disadvantages

- Advantages
  - Decoupled
  - Simpler than state monad
- Disadvantages
  - Extra boilerplate needed

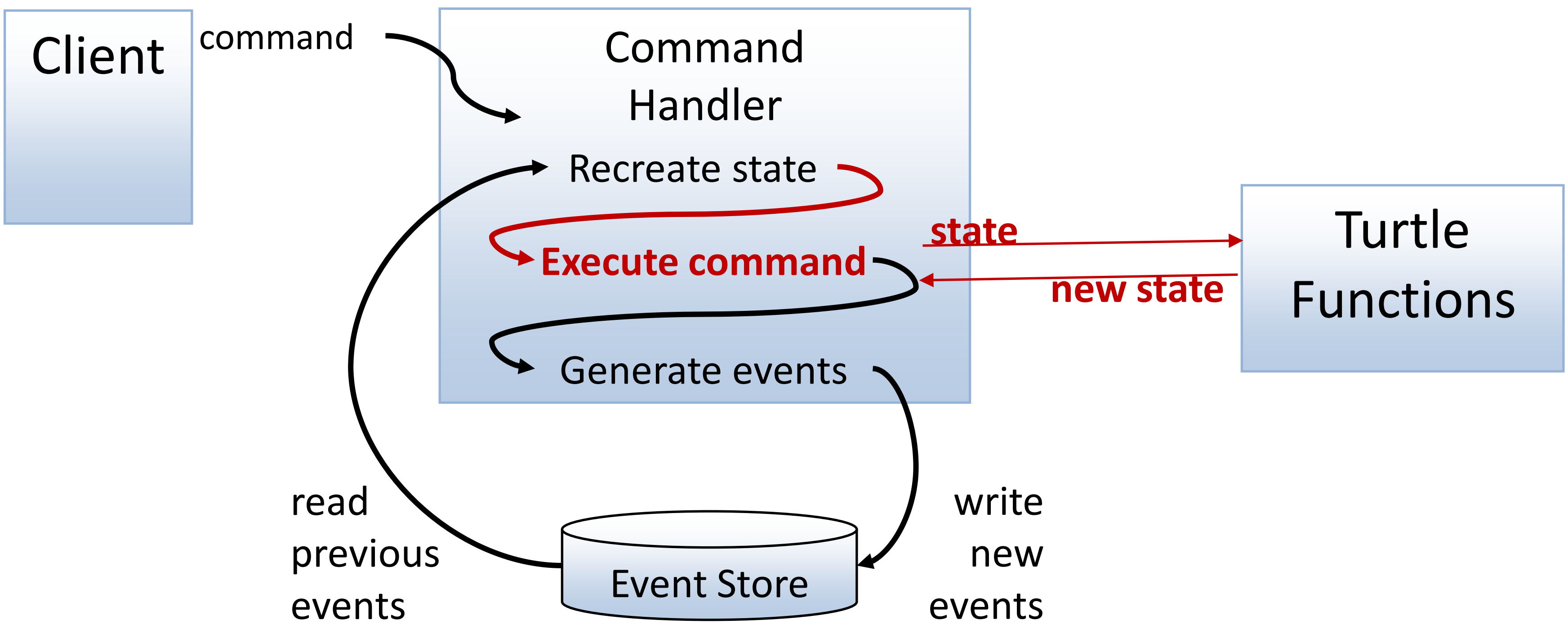
8. Event Sourcing

How should we persist state?

Overview

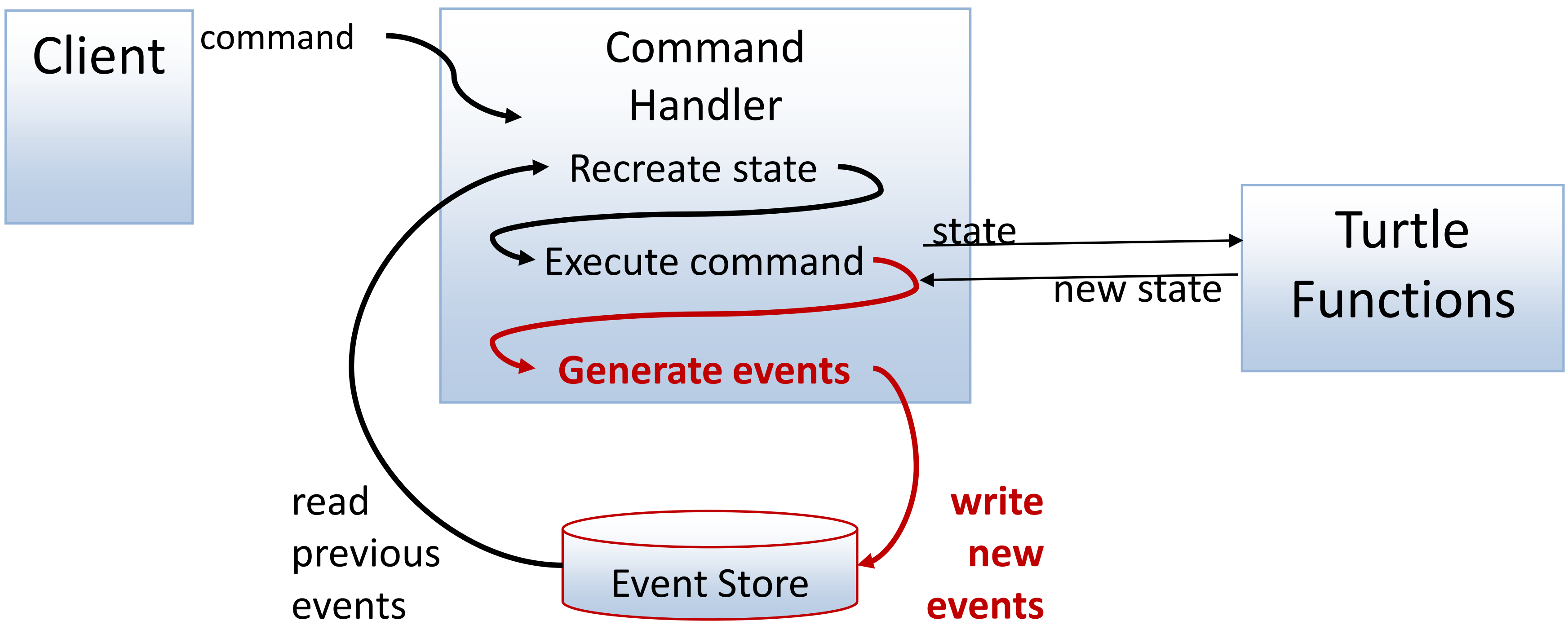

Client
command
Command
Handler
Recreate state
Execute command
Generate events
state
new state
Turtle
Functions
read
previous
events
Event Store
write
new
events

Overview



Client
command
Command
Handler
Recreate state
Execute command
Generate events
state
new state
Turtle
Functions
read
previous
events
Event Store
write
new
events

Overview


Client
command
Command Handler
Recreate state
Execute command
Generate events
state
new state
Turtle Functions
read previous events
Event Store
write new events

Command vs. Event

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
```

What you *want* to have happen

Command vs. Event

```
type TurtleEvent =
    | Moved of Distance * StartPosition * EndPosition
    | Turned of AngleTurned * FinalAngle
    | PenStateChanged of PenState
```

What *actually* happened (past tense)

Compare with the command

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
```

Implementation

```
/// Apply an event to the current state and return the new state
let applyEvent state event =
    match event with
    | Moved (distance,startPosition,endPosition) ->
        {state with position = endPosition }  // don't call Turtle
    | Turned (angleTurned,finalAngle) ->
        {state with angle = finalAngle}
    | PenStateChanged penState ->
        {state with penState = penState}
```

Important!
No side effects when recreating state!

Implementation

```
let handleCommand (command:TurtleCommand) =
    // First load all the events from the event store
    let eventHistory = EventStore.getEvents()
    // Then, recreate the state before the command
    let stateBeforeCommand =
        eventHistory |> List.fold applyEvent Turtle.initialTurtleState
    // Create new events from the command
    let newEvents =
        executeCommand command stateBeforeCommand
    // Store the new events in the event store
    events |> List.iter (EventStore.saveEvent)
```

No side effects

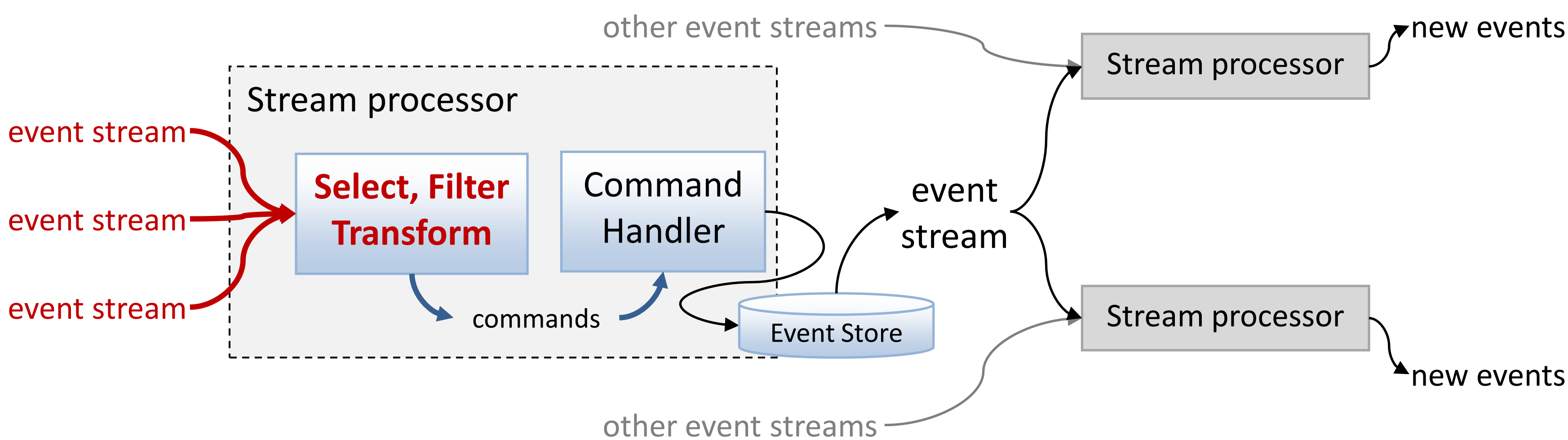
This is where side-effects happen

Advantages and disadvantages

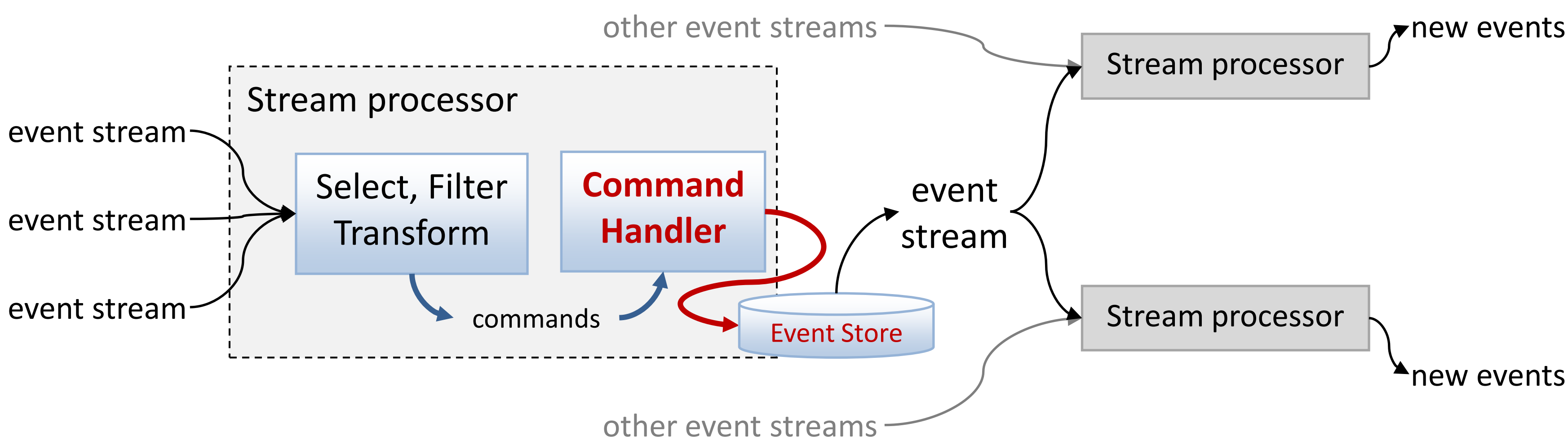
- Advantages
  - Decoupled
  - Stateless
  - Supports replay of events
- Disadvantages
  - More complex
  - Versioning

9. Stream Processing

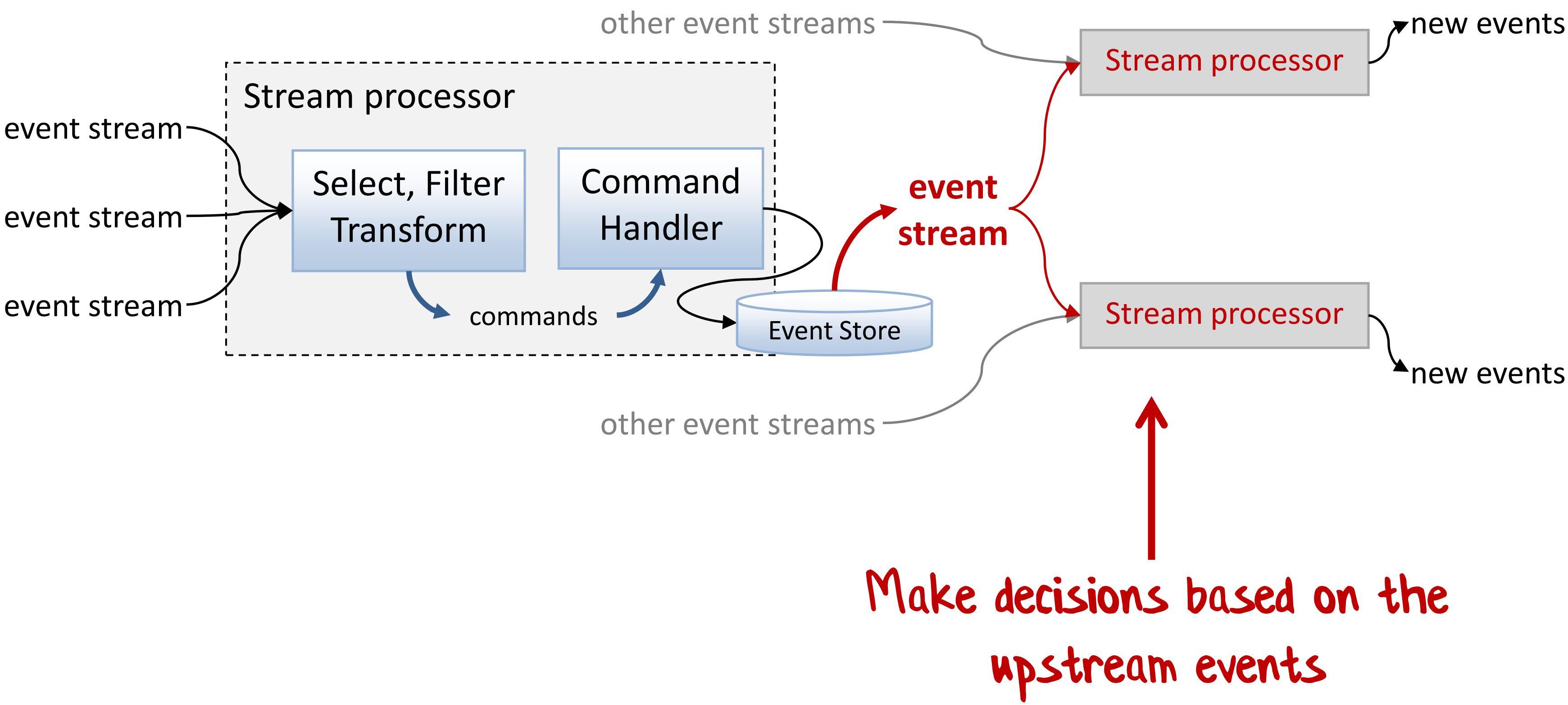
Overview


other event streams
new events
Stream processor
Stream processor
event stream
event stream
event stream
Select, Filter
Transform
Command
Handler
commands
Event Store
event
stream
Stream processor
new events
other event streams

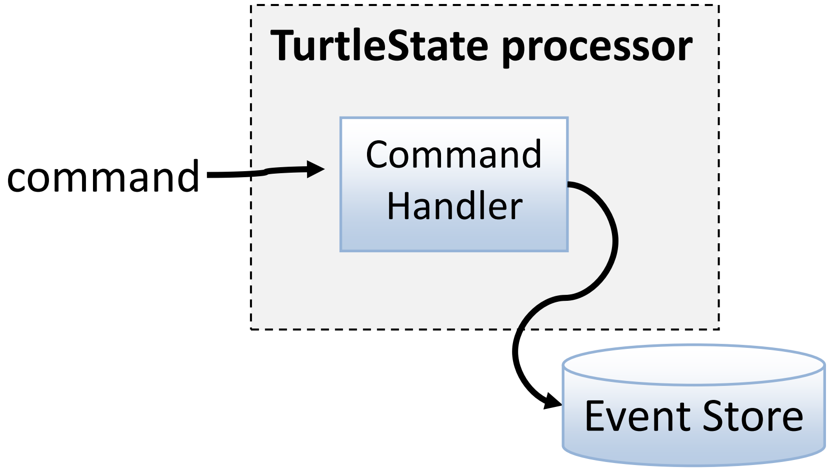
Overview


other event streams
new events
Stream processor
Stream processor
event stream
event stream
event stream
Select, Filter
Transform
Command
Handler
commands
Event Store
event
stream
Stream processor
new events
other event streams

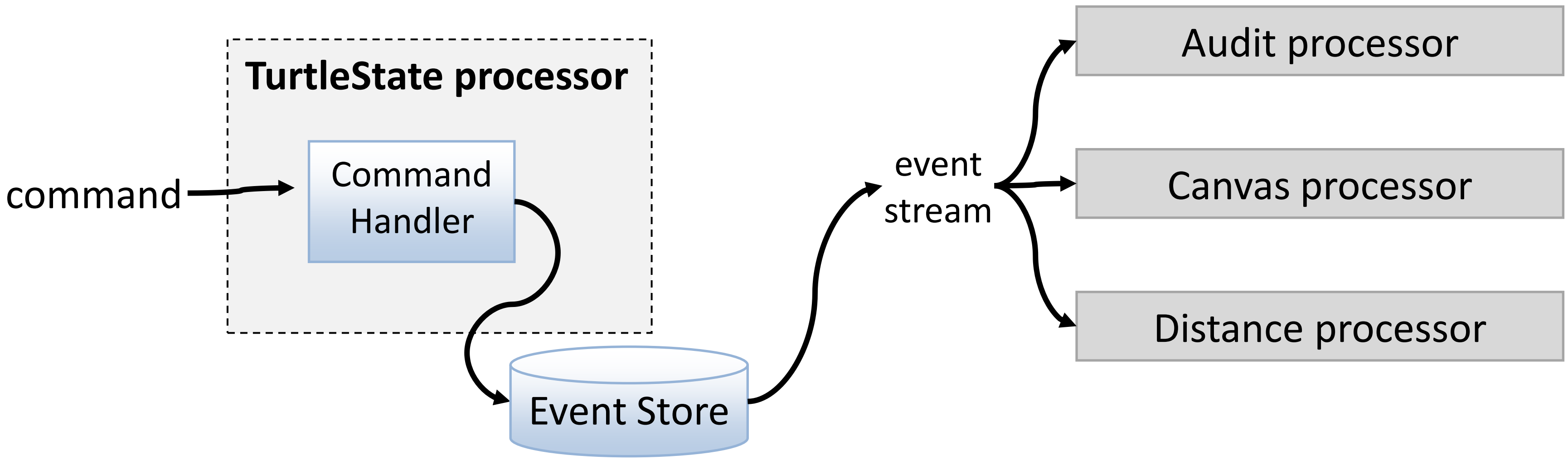
Overview


other event streams
new events
Stream processor
Stream processor
event stream
event stream
event stream
Select, Filter Transform
Command Handler
commands
Event Store
event stream
Stream processor
new events
other event streams
Make decisions based on the upstream events

Turtle stream processing example



Turtle stream processing example



Stream processing demo

Auditing Processor

Canvas Processor

Distance Travelled Processor

Advantages and disadvantages

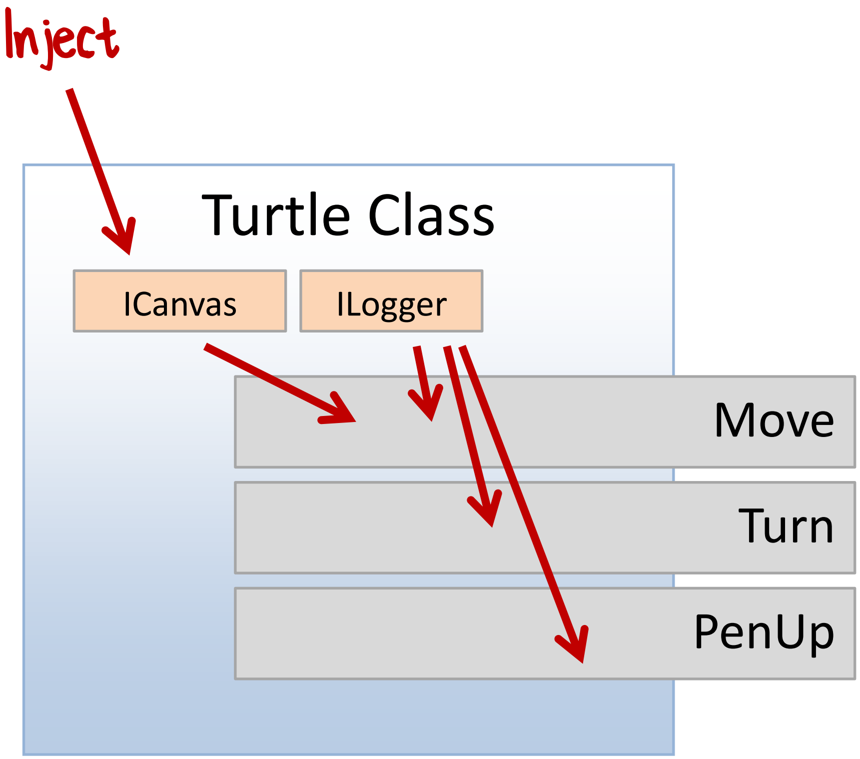
- Advantages
  - Same as Event Sourcing, plus
  - Separates state logic from business logic
  - Microservice friendly!
- Disadvantages
  - Even more complex!

Review

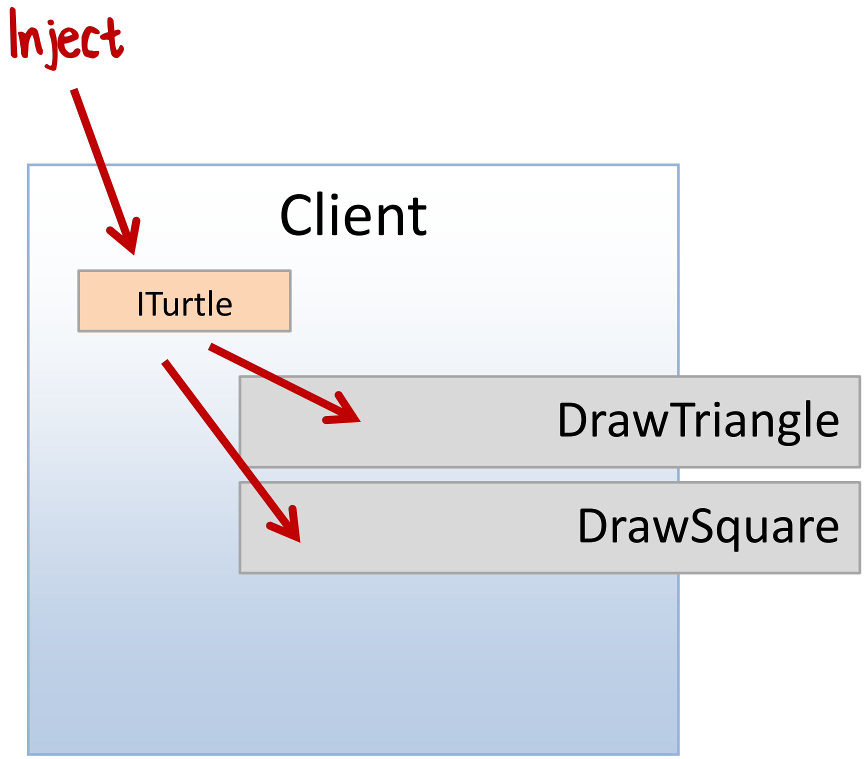
- "Conscious decoupling" with data types
  - Passing data instead of calling functions
- Immutable data stores
  - Storing event history rather than current state

10. OO style dependency injection

Overview



Same for the Turtle client



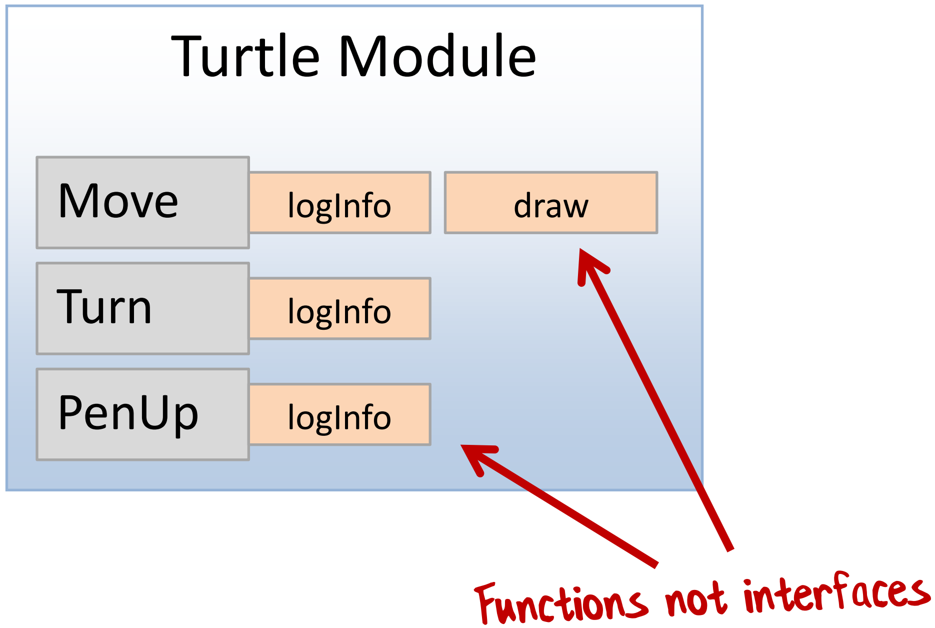
OO dependency injection demo

Advantages and disadvantages

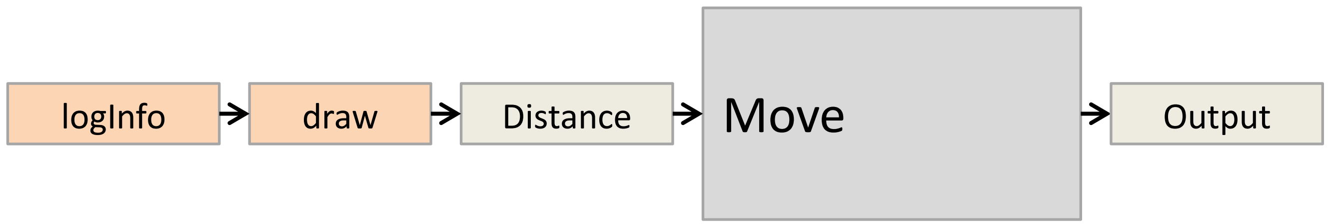
- Advantages
  – Well understood
- Disadvantages
  – Unintentional dependencies
  – Interfaces often not fine grained
  – Often requires IoC container or similar

11. FP style dependency injection

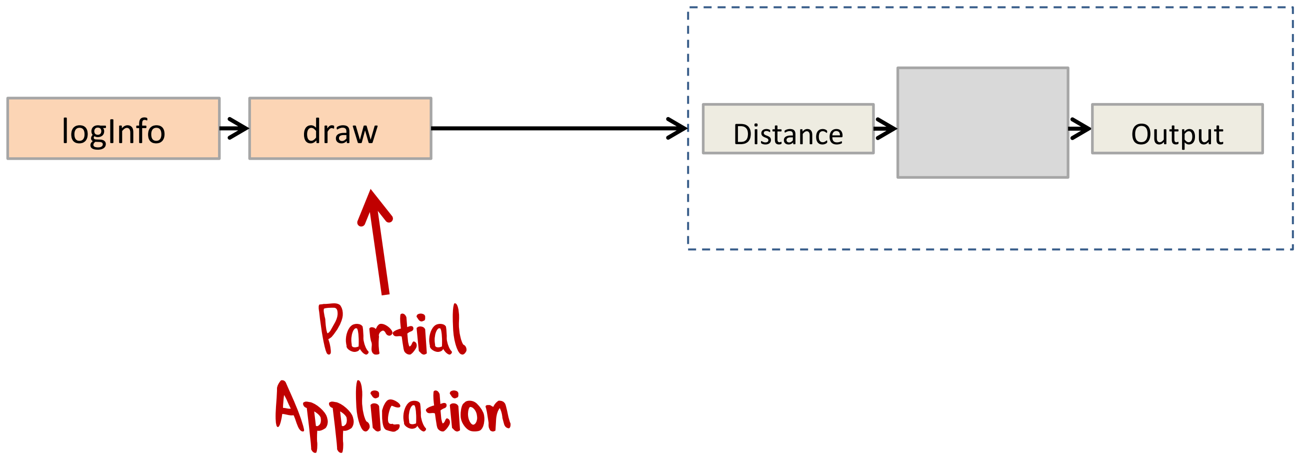
Overview


Turtle Module
Move
logInfo
draw
Turn
logInfo
PenUp
logInfo
Functions not interfaces

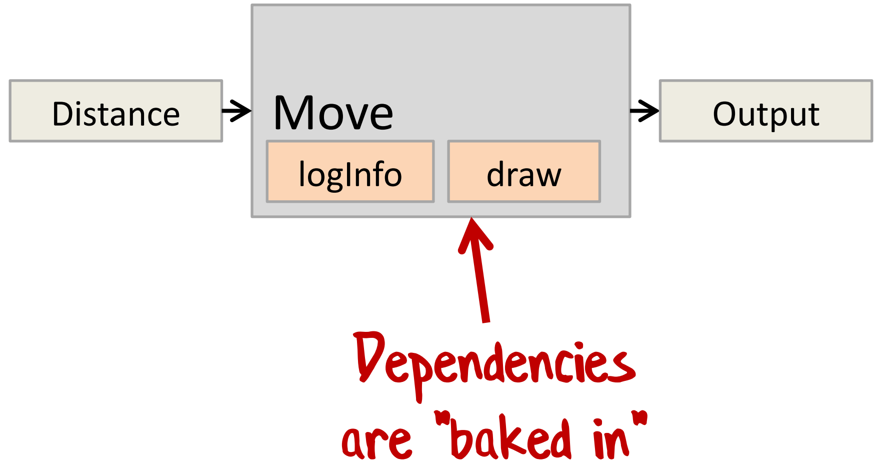
Overview



Overview



Overview


Distance
Move
logInfo
draw
Output
Dependencies
are "baked in"

Function parameters

"move" implementation

```
let move logInfo draw distance state =
    logInfo (sprintf "Move %0.1f" distance)
    // calculate new position
    ...
    // draw line if needed
    if state.penState = Down then
        draw startPosition endPosition
    // update the state
    ...
```

Function parameter

"turn" implementation

```
let turn logInfo angleToTurn state =
    logInfo (sprintf "Turn %0.1f" angleToTurn)
    // calculate new angle
    let newAngle = ...
    // update the state
    ...
```

Partial application in practice

```
let move = Turtle.move Logger.info Canvas.draw
// output is new function: "move"
//     (Distance -> TurtleState -> TurtleState)

let turn = Turtle.turn Logger.info
// output is new function: "turn"
//     (float<Degrees> -> TurtleState -> TurtleState)
```

partial application

```
Turtle.initialTurtleState
|> move 50.0
|> turn 120.0<Degrees>
```

Use new functions here

FP dependency injection demo

Advantages and disadvantages

- Advantages
  - Dependencies are explicit
  - Functions, not interfaces
  - Counterforce to having too many dependencies (ISP for free!)
  - Built in! No special libraries needed
- Disadvantages
  - ??

12. Interpreter

APIs create coupling

```
module TurtleAPI =
    move : Distance -> State -> Distance * State
    turn : Angle -> State -> State
    penUp : State -> State
    penDown : State -> State
```

Fine, but what if the API
changes to return Result?

APIs create coupling

```
module TurtleAPI =
   move : Distance -> State -> Result<Distance * State>
   turn : Angle -> State -> Result<State>
   penUp : State -> Result<State>
   penDown : State -> Result<State>
```

Fine, but what if it needs to be Async as well?

APIs create coupling

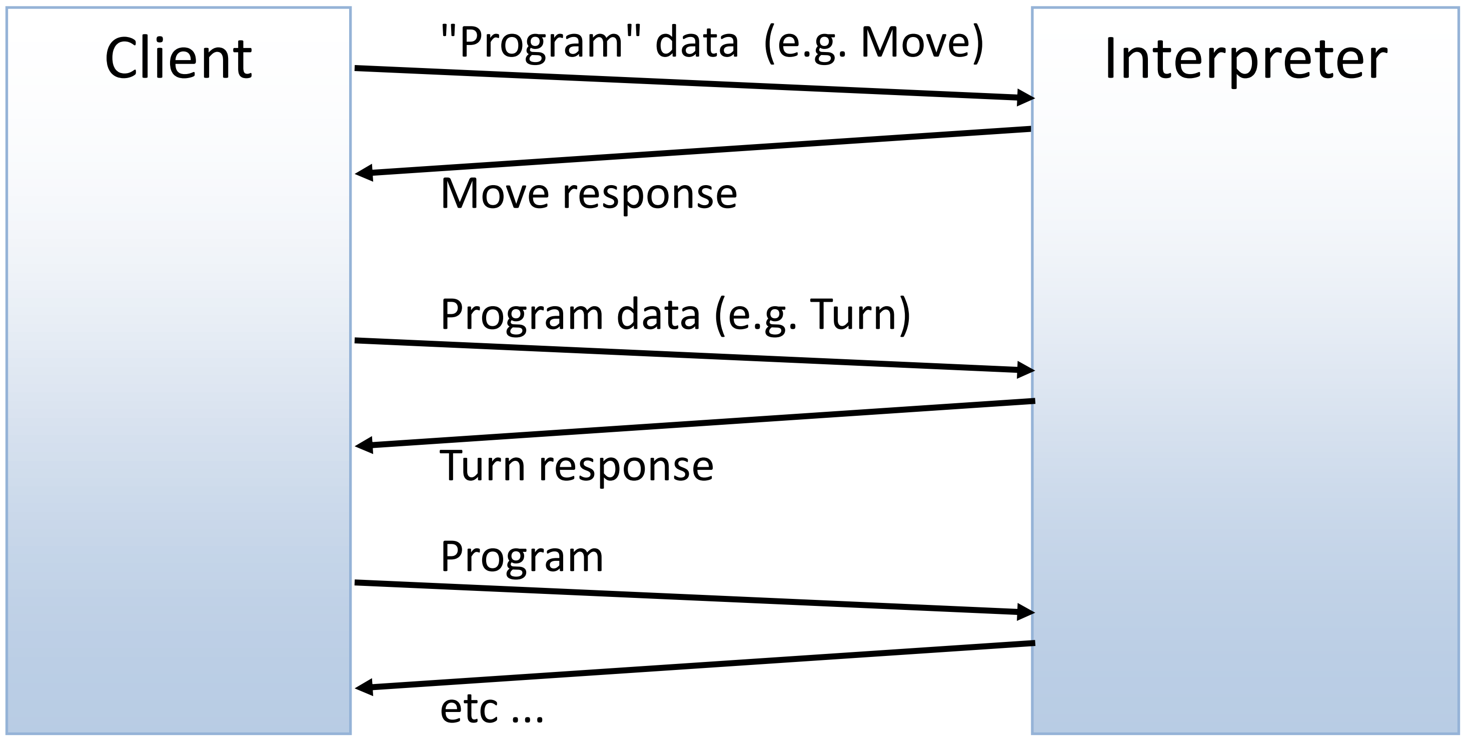
```
module TurtleAPI =
   move : Distance -> State -> AsyncResult<Distance * State>
   turn : Angle -> State -> AsyncResult<State>
   penUp : State -> AsyncResult<State>
   penDown : State -> AsyncResult<State>
```

Each change breaks the caller ☹

Solution: decouple using data instead of functions!

But how to manage control flow?

Overview


Client
"Program" data (e.g. Move)
Interpreter
Move response
Program data (e.g. Turn)
Turn response
Program
etc ...

Create a "Program" type

```
type TurtleProgram =
    //          (input params)   (response)
    | Move      of Distance    * (Distance -> TurtleProgram)
    | Turn      of Angle       * (unit -> TurtleProgram)
    | PenUp     of (* none *)    (unit -> TurtleProgram)
    | PenDown   of (* none *)    (unit -> TurtleProgram)
    | Stop
```

New case needed!

Input

Output from interpreter

Next step for the interpreter

Usage example

```
let drawTriangle =
    Move (100.0, fun actualDistA ->
                    // actualDistA is response from interpreter
    Turn (120.0<Degrees>, fun () ->
                    // () is response from interpreter
    Move (100.0, fun actualDistB ->
    Turn (120.0<Degrees>, fun () ->
    Move (100.0, fun actualDistC ->
    Turn (120.0<Degrees>, fun () ->
    Stop))))))
```

Ugly!

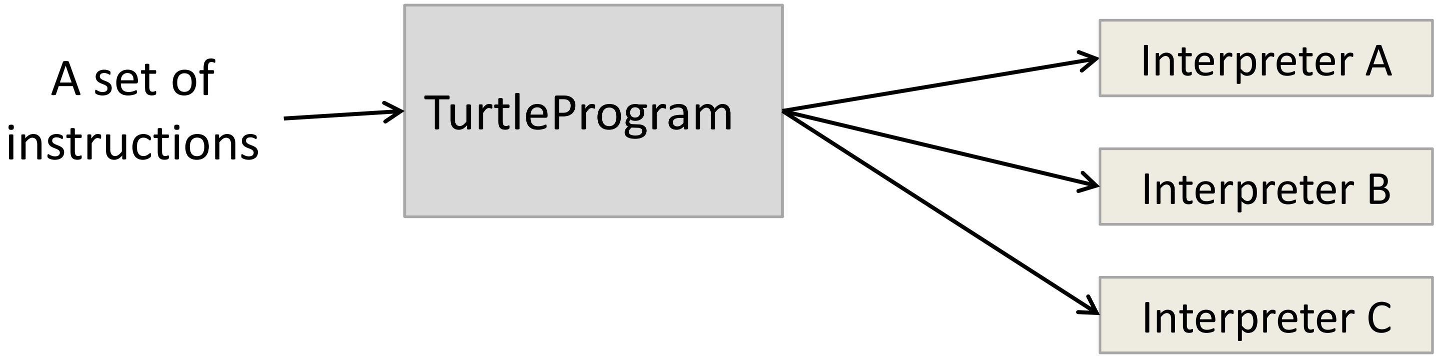
Can we hide the continuations behind the scenes?

Yes, we can!

Usage example

```
let drawTriangle = turtleProgram {       // Seen this trick before!
    let! actualDistA = move 100.0
    do! turn 120.0<Degrees>
    let! actualDistB = move 100.0
    do! turn 120.0<Degrees>
    let! actualDistC = move 100.0
    do! turn 120.0<Degrees>
    }
```

Overview


A set of
instructions
TurtleProgram
Interpreter A
Interpreter B
Interpreter C

Example: Turtle Interpreter

```
let rec interpretAsTurtle state program =
    match program  with
    | Stop ->
        state
    | Move (dist, next) ->
        let actualDistance, newState = Turtle.move dist state
        let nextProgram = next actualDistance // next step
        interpretAsTurtle newState nextProgram
    | Turn (angle, next) ->
        let newState = Turtle.turn angle state
        let nextProgram = next()       // next step
        interpretAsTurtle newState nextProgram
```

Move the turtle

Execute the next step

Example: Distance Interpreter

```
let rec interpretAsDistance distanceSoFar program =
   match program with
   | Stop ->
      distanceSoFar
   | Move (dist, next) ->
      let newDistance = distanceSoFar + dist
      let nextProgram = next newDistance
      interpretAsDistance newDistance nextProgram
   | Turn (angle, next) ->
      // no change in distanceSoFar
      let nextProgram = next()
      interpretAsDistance distanceSoFar nextProgram
```

Calculate the new distance

Execute the next step

Interpreter demo

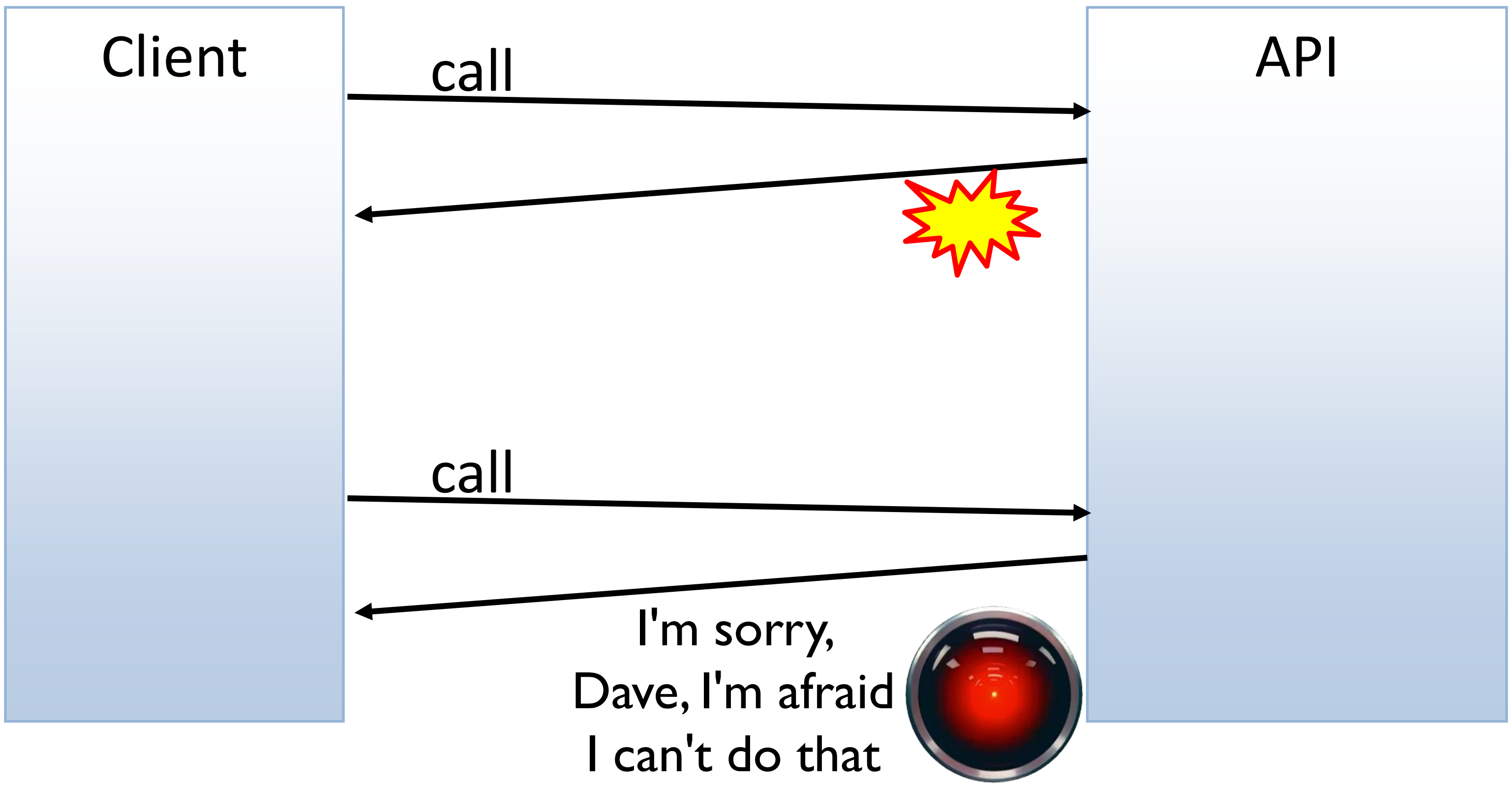
Advantages and disadvantages

- Advantages
  - Completely decoupled
  - Pure API
  - Optimization possible
- Disadvantages
  - Complex
  - Best with limited set of operations

Examples: Twitter's "Stitch" library, Facebook's "Haxl"

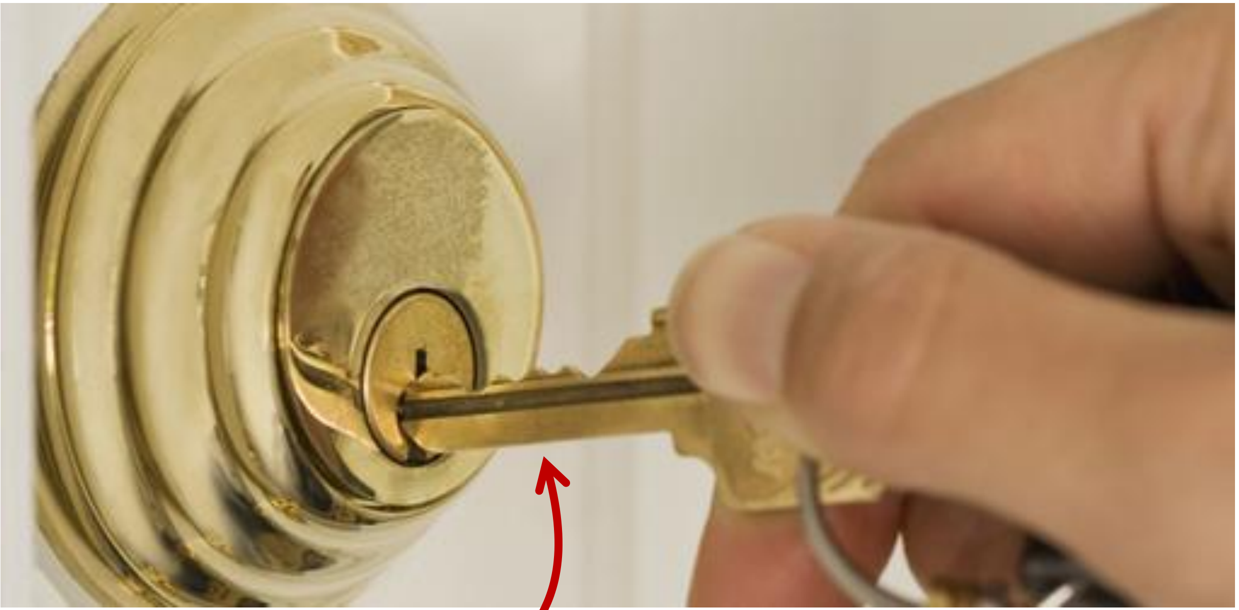
13. Capabilities

Overview



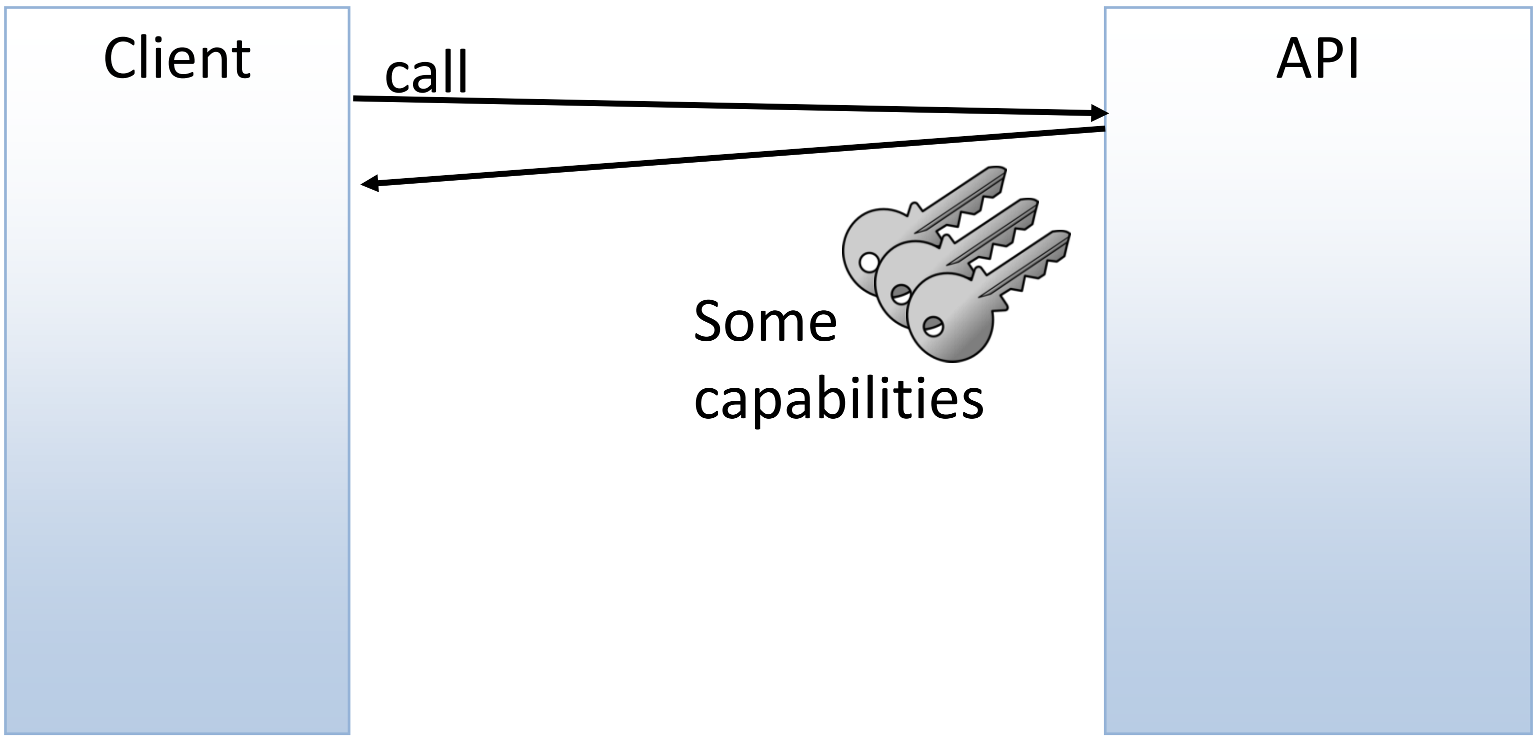
Rather than telling me what I can't do,
why not tell me what I can do?

Capability-based API

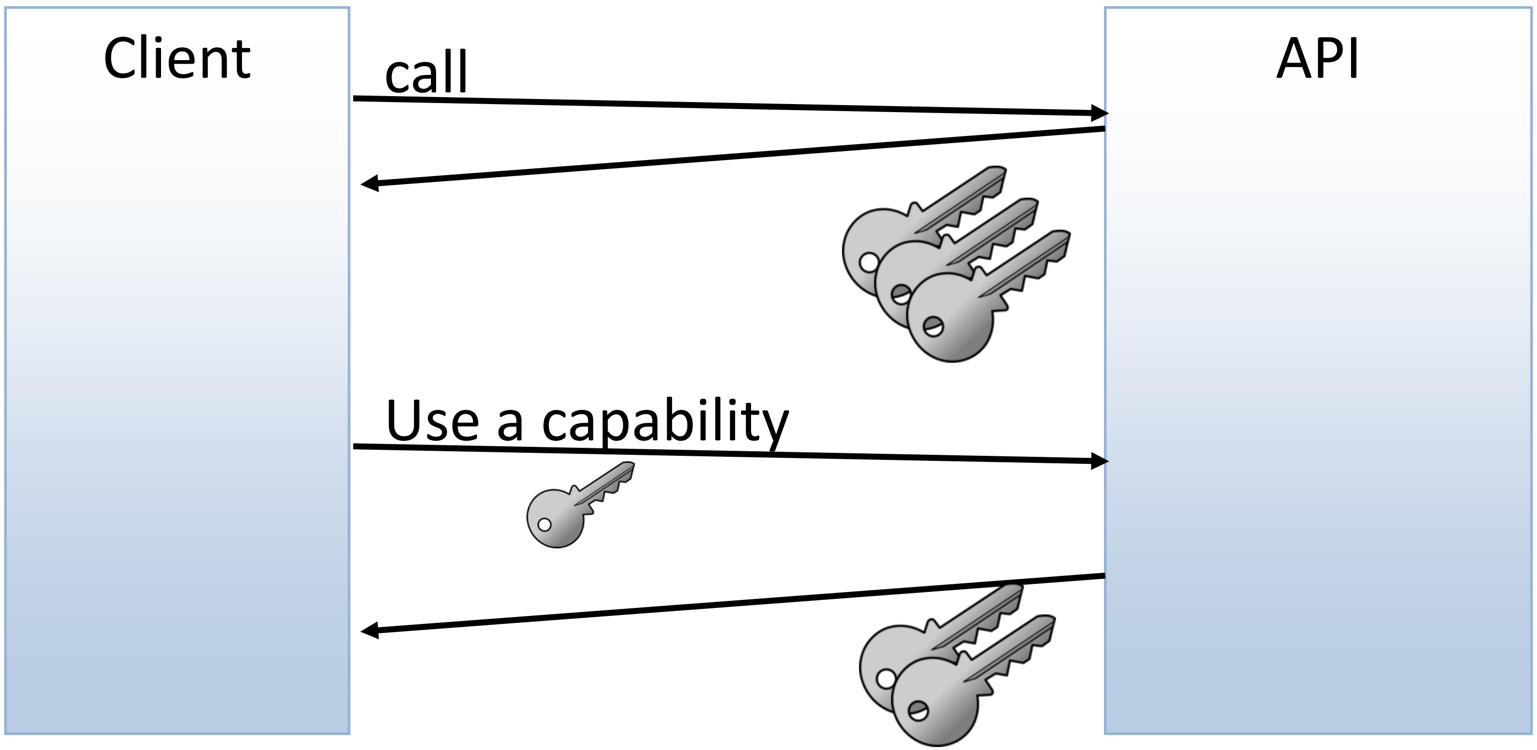


A capability

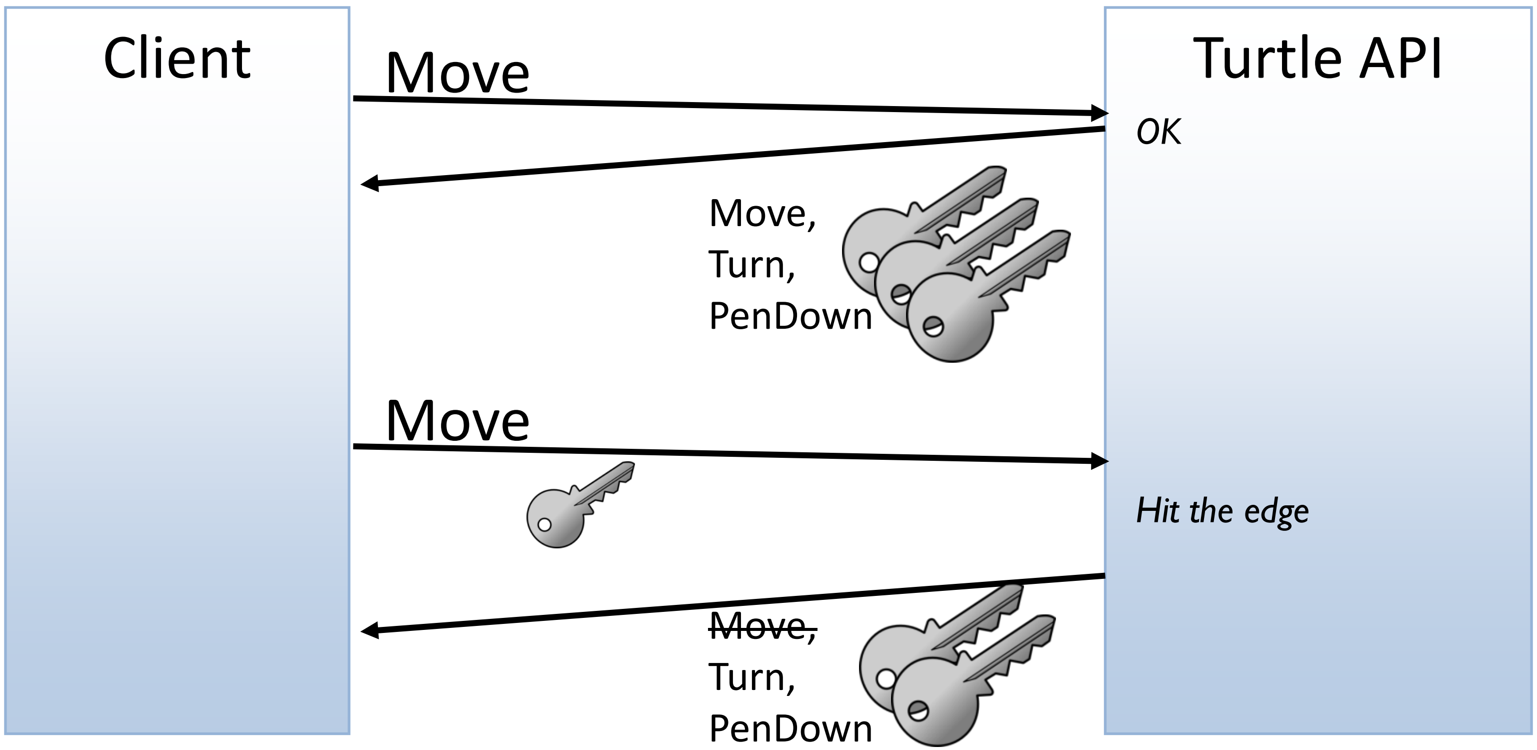
Overview



Overview



Turtle API capabilities



Turtle Capabilities

```
type TurtleCapabilities = {
    move    : MoveFn option
    turn    : TurnFn
    penUp   : PenUpDownFn
    penDown: PenUpDownFn
    }
and MoveFn =  Distance -> TurtleCapabilities
and TurnFn =     Angle -> TurtleCapabilities
and PenUpDownFn = unit -> TurtleCapabilities
```

Usage example

```
let turtleCaps = Turtle.start()      // initial set of capabilities
match turtleCaps.move with
| None ->
    warn "Error: Can't do move 1"
    turtleCaps.turn 120<Degrees>
| Some moveFn ->      // OK
    let turtleCaps = moveFn 60.0   // a new set of capabilities
    match turtleCaps.move with
    | None ->
        warn "Error: Can't do move 2"
        turtleCaps.turn 120<Degrees>
    | Some moveFn ->
        ...
```

Capabilities demo

Advantages and disadvantages

- Advantages
  - Client doesn't need to duplicate business logic
  - Better security
  - Capabilities can be transformed for business rules
- Disadvantages
  - Complex to implement
  - Client has to handle unavailable functionality

Examples: HATEOAS

Phew!

fsharpforfunandprofit.com/turtle

Slides and video here

F# consulting

fsharpWorks

More F# at fsharp.org

fsharp.org