Evolution Engineering

Functional streams with FS2

Download the workshop files at:
github.com/evolution-gaming/fs2-workshop

$ whoami

Name: Raitis Rusins Krikis (Just call me Raity)

LinkedIn: linkedin.com/in/rusins

Occupation: Scala developer at Evolution for 3+ years
based in Riga, Latvia

Interests: Programming, free open source software,
building electric skateboards

DOWNLOAD THE WORKSHOP FILES!

github.com/evolution-gaming/fs2-workshop

What are we going to do today?

We will:

- Learn some FS2 basics
- Solve interesting problems with FS2

We won’t be:

- Covering effect monads
- Getting technical about how FS2 works
- Learning best practices

DOWNLOAD THE WORKSHOP FILES!

github.com/evolution-gaming/fs2-workshop

Before we begin

- Download the workshop exercises: github.com/evolution-gaming/fs2-workshop
- Import the SBT project into your favorite IDE. Or just open the source files in Emacs, Vim, Evernote, pick your poison.
- Run SBT in the project folder and compile to make sure all the dependencies are downloaded.
- You can run the exercises with run within SBT

Pure Streams

• Stream.empty	// Stream()
• Stream(1, 2, 3)	// Stream(1, 2, 3)
• Stream.emits(Seq(1, 2, 3))	// Stream(1, 2, 3)
• Stream(1, 2) ++ Stream(3)	// Stream(1, 2, 3)
• Stream(1, 2, 3).map(_ + 1).filter(_ % 2 == 0)	// Stream(2, 4)
• Stream(“cave”, “echo”).flatMap(word => Stream(word.capitalize, word))	// Stream(“Cave”, “cave”, “Echo”, “echo”)
• Stream(1, 2).repeat.take(5)	// Stream(1, 2, 1, 2, 1)
• .toVector and .toList	Convert pure streams into Scala collections

Task 1 – The Collatz Conjecture

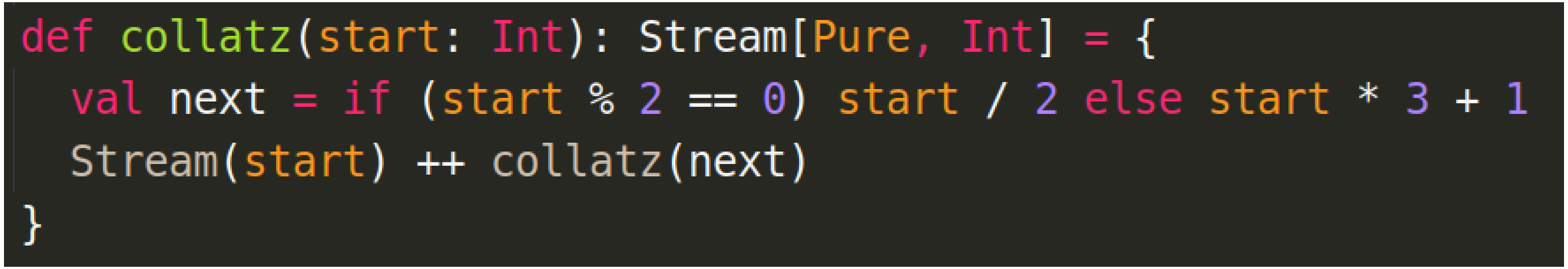
The objective is to model an infinite sequence of positive integers, where each one is obtained from the previous one.

If the previous number X is even, then the next number will be X / 2.
If the previous number X is odd, then the next number will be 3X + 1.
The starting number is given.

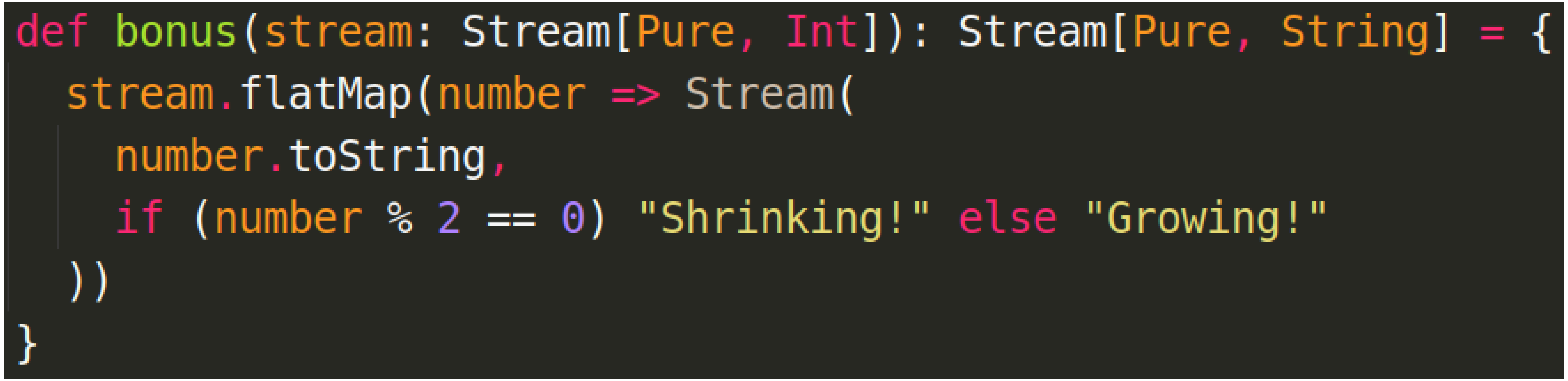
To achieve this, you will have to use a recursive function!

Since streams are evaluated lazily, this is safe to do :)

Task 1 Solution



```scala
def collatz(start: Int): Stream[Pure, Int] = {
  val next = if (start % 2 == 0) start / 2 else start * 3 + 1
  Stream(start) ++ collatz(next)
}
```



```scala
def bonus(stream: Stream[Pure, Int]): Stream[Pure, String] = {
  stream.flatMap(number => Stream(
    number.toString,
    if (number % 2 == 0) "Shrinking!" else "Growing!"
  ))
}
```

Effectful Streams

- Stream.eval(IO(println(“Hi!”)))

  // Stream[IO, Unit], prints Hi! whenever evaluated

- Stream(1, 2, 3).evalMap(x => IO(
    println(s“Evaluating $x”)
    x
  ))

  // Stream[IO, Int], returns values 1, 2, 3, and prints the numbers when evaluated

- val a = Stream(‘1’, ‘2’, ‘3’).evalMap(IO(_))
  val b = Stream(‘a’, ‘b’, ‘c’).evalMap(IO(_))
  a merge b

  // Stream[IO, Char], returns a stream with both input streams merged in a non-deterministic manner!

  They are evaluated asynchronously, whichever stream returns an element first is the one returned.

  Example outputs:

  123abc, abc123, a123bc, 12a3bc, a1b2c3, ...

Task 2 – Asynchronous FooBar

You are given three input streams:

- count – An infinite stream that prints increasing integers as its effect. 1,2,3,…
- foos – An infinite stream that prints “foo” as its effect. foo, foo, foo…
- bars – An infinite stream that prins “bar” as its effect. bar, bar, bar…

With the help of methods for effectful streams (merge, delayBy, metered, interruptAfter), your objective is to use these three input streams to implement the standard FooBar “interview” task:

„Write a program to print 15 lines of output. On each line, print the line number. For each line, if the line number is divisible by 3, print “foo”. If the line number is divisible by 5, print “bar”.

Task 2 Solution

```
private val stream = {
  val numbers = count.metered(1.second)
  val foo = foos.metered(3.seconds)
  val bar = bars.metered(5.seconds)
  val foobar = foo merge bar.delayBy(100.millis)

  (numbers merge foobar.delayBy(100.millis)).interruptAfter(15.seconds + 500.millis)
}
```

Concurrent Streams

• streamA.concurrently(streamB)	// Runs streamB in the background. StreamA terminating will terminate streamB, but not the other way round.
• SignallingRef[IO, X](initialValue)	// IO[SignallingRef[IO, X]] – a reference to a mutable, atomic, thread safe value of type X
• signal.get()	// IO[X] – gets the current value of the signal
• signal.set(x)	// IO[Unit] – sets the value of the signal

Task 3 – Red Light, Green Light

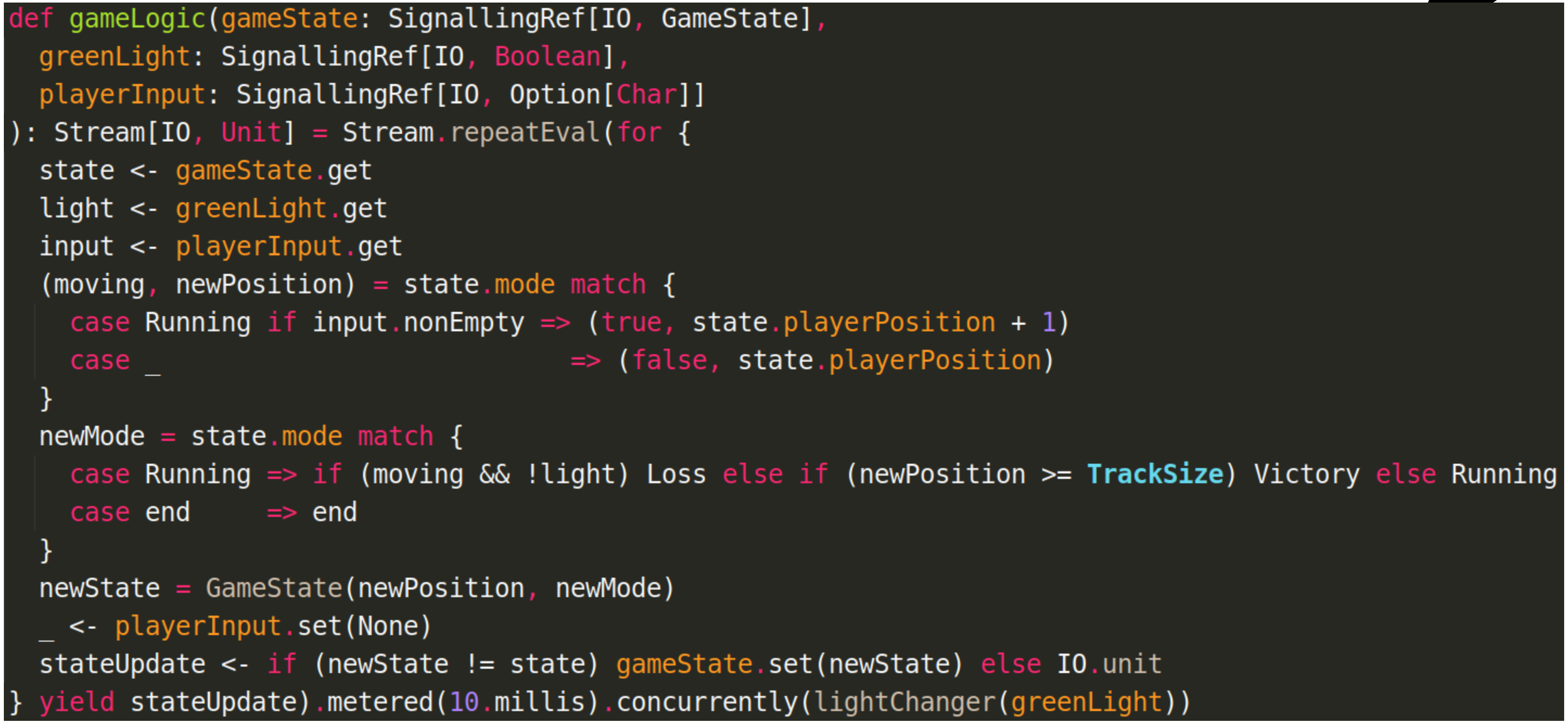
Red Light Green Light is a game in which players are only allowed to move when when it is “green light”. If a player moves when it is “red light”, they lose. The objective for the players is to get from point A to point B.

In this task, you will be implementing the game logic for such a game using streams and signals!

You will have to implement a function that is given signals for the current player input, current light state, and current game state, and will have to return a stream that concurrently updates the light signal, as well as concurrently updates the game state based on the rules! You should meter the light updating stream to happen every 2 seconds, and meter the game state updating stream to 10 milliseconds.

Tip: Look around the existing code to see how things are done!

Task 3 Solution



```scala
def gameLogic(gameState: SignallingRef[IO, GameState],
  greenLight: SignallingRef[IO, Boolean],
  playerInput: SignallingRef[IO, Option[Char]]
): Stream[IO, Unit] = Stream.repeatEval(for {
  state <- gameState.get
  light <- greenLight.get
  input <- playerInput.get
  (moving, newPosition) = state.mode match {
    case Running if input.nonEmpty => (true, state.playerPosition + 1)
    case _                           => (false, state.playerPosition)
  }
  newMode = state.mode match {
    case Running => if (moving && !light) Loss else if (newPosition >= TrackSize) Victory else Running
    case end     => end
  }
  newState = GameState(newPosition, newMode)
  _ <- playerInput.set(None)
  stateUpdate <- if (newState != state) gameState.set(newState) else IO.unit
} yield stateUpdate).metered(10.millis).concurrently(lightChanger(greenLight))
```

Statefully transforming streams

Problem: we need a way of looking at one or more elements of a stream, and then having the ability to stop, continue, process another stream, or anything in between. All while being able to access and mutate some state we carry along. (Without relying on concurrency primitives, lol.)

Streams are evaluated lazily, so this process needs to be expressed lazily as well.

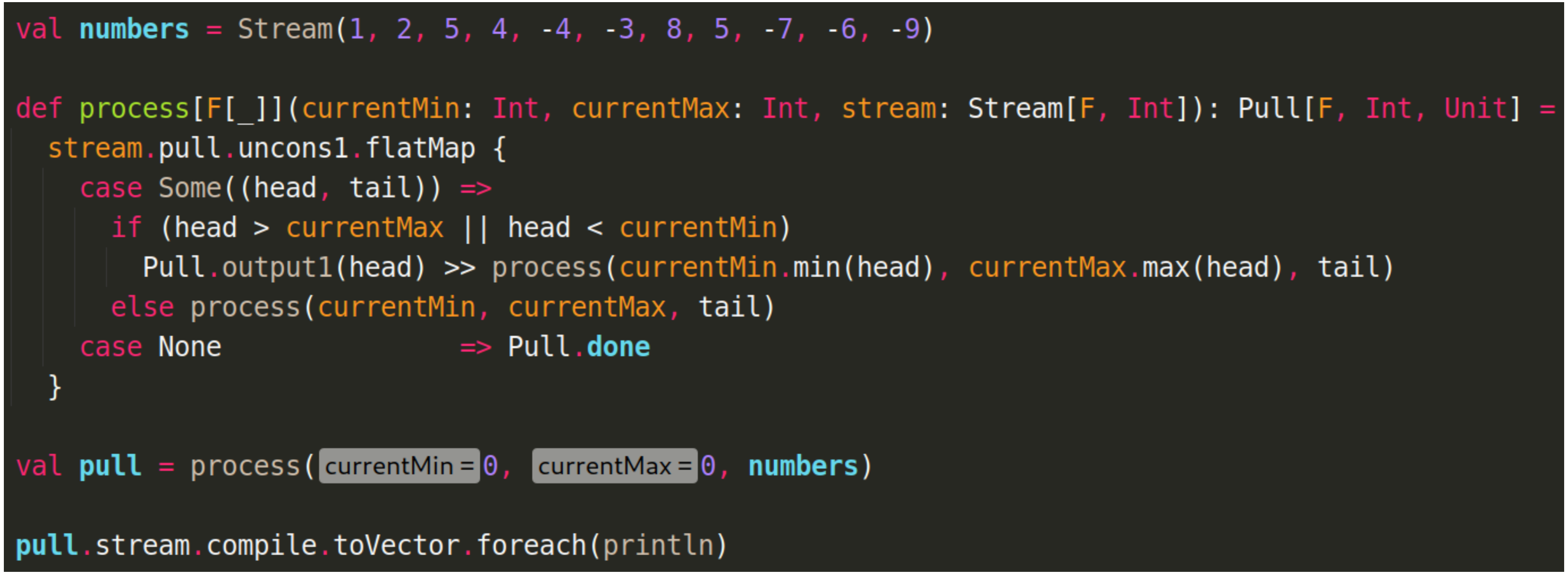
pull: Pull[+F[_], +O, +R]

- F[_] – The effect type (IO, Task, etc.)
- O – The resulting stream output type. What we care about!
- R – Intermediate result type, used for flatMapping. Often Unit.

- stream.pull.uncons1: Pull[F, Nothing, Option[(X, Stream[F, X])]]
  turns a stream into a pull where we try to take one element and
  the remaining stream. Use flatMap to access the element and tail.
- Pull.output1(x) – a pull that outputs element x to the output stream.
- Pull.done – a pull that has finished processing.
- stream.pull.echo – a pull that outputs all elements of the stream.
- pullA >> pullB – constructs a pull that first runs pullA, and then pullB.
- pull.stream – Turns a pull back into a stream.

Pull example

Filtering a stream of numbers to only return values that are new maximums or minimums. Prints “1, 2, 5, -4, 8, -7, -9”



```scala
val numbers = Stream(1, 2, 5, 4, -4, -3, 8, 5, -7, -6, -9)

def process[F[_]](currentMin: Int, currentMax: Int, stream: Stream[F, Int]): Pull[F, Int, Unit] =
  stream.pull.uncons1.flatMap {
    case Some((head, tail)) =>
      if (head > currentMax || head < currentMin)
        Pull.output1(head) >> process(currentMin.min(head), currentMax.max(head), tail)
      else process(currentMin, currentMax, tail)
    case None               => Pull.done
  }

val pull = process(currentMin = 0, currentMax = 0, numbers)

pull.stream.compile.toVector.foreach(println)
```

Task 4 – Chat logs

In this task you will be merging two streams into a single one using pulls!

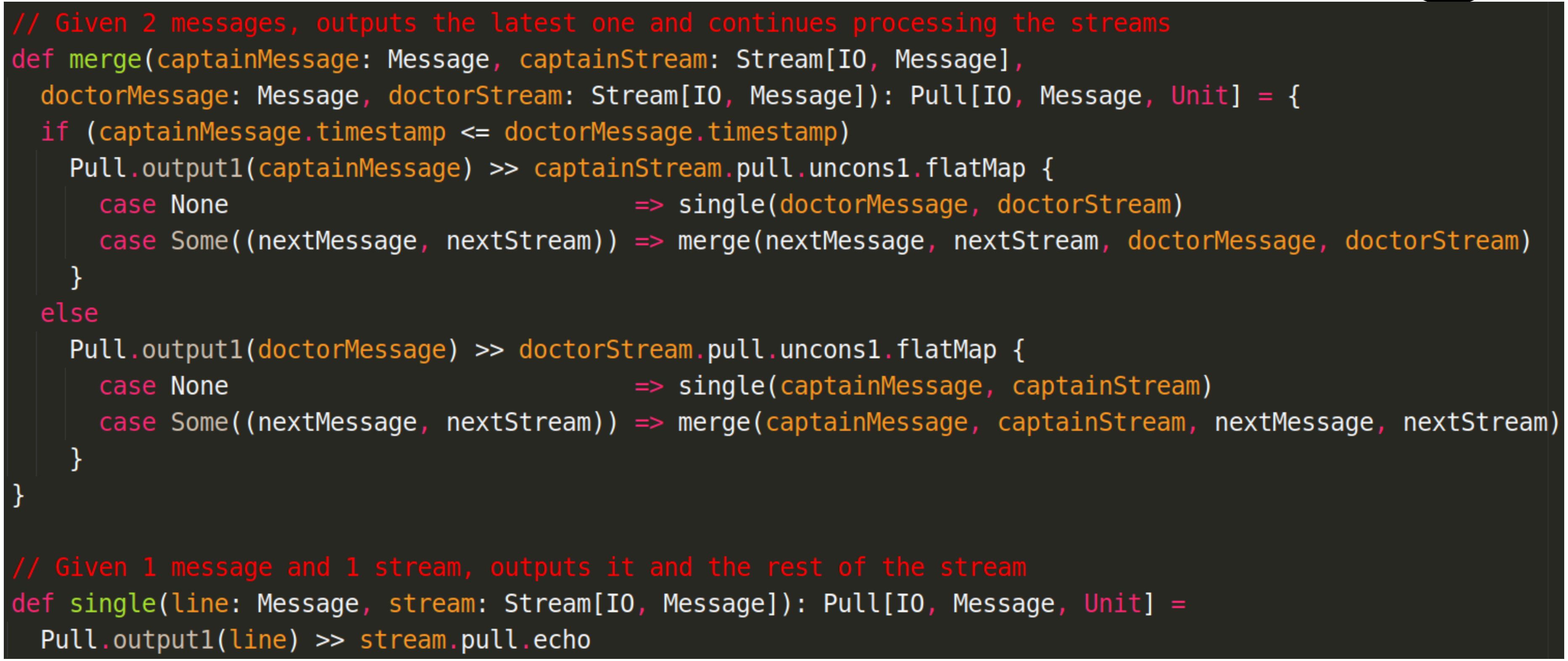
You are given two streams of messages in chronological order from a person: captain and doctor.
Each message object contains a timestamp field by which you should decide which message should be outputted first in the output stream.

The input streams are read from assets/captain.txt and assets/doctor.txt for you already. You can take a look if you are curious what the correct output should look like. You can also look at PullExample.scala

Task 4 Solution pt.1

```scala
val firstPull: Pull[IO, Message, Unit] = for {
  c <- captainFileStream.pull.uncons1
  d <- doctorFileStream.pull.uncons1
  action <- (c, d) match {
    case (None, None)                                                                 =>
      Pull.done
    case (Some((captainMessage, captainStream)), Some((doctorMessage, doctorStream))) =>
      merge(captainMessage, captainStream, doctorMessage, doctorStream)
    case (Some((captainMessage, captainStream)), None)                                =>
      single(captainMessage, captainStream)
    case (None, Some((doctorMessage, doctorStream)))                                  =>
      single(doctorMessage, doctorStream)
  }
} yield action
```

Task 4 Solution pt.2



```scala
// Given 2 messages, outputs the latest one and continues processing the streams
def merge(captainMessage: Message, captainStream: Stream[IO, Message],
  doctorMessage: Message, doctorStream: Stream[IO, Message]): Pull[IO, Message, Unit] = {
  if (captainMessage.timestamp <= doctorMessage.timestamp)
    Pull.output1(captainMessage) >> captainStream.pull.uncons1.flatMap {
      case None                           => single(doctorMessage, doctorStream)
      case Some((nextMessage, nextStream)) => merge(nextMessage, nextStream, doctorMessage, doctorStream)
    }
  else
    Pull.output1(doctorMessage) >> doctorStream.pull.uncons1.flatMap {
      case None                           => single(captainMessage, captainStream)
      case Some((nextMessage, nextStream)) => merge(captainMessage, captainStream, nextMessage, nextStream)
    }
}

// Given 1 message and 1 stream, outputs it and the rest of the stream
def single(line: Message, stream: Stream[IO, Message]): Pull[IO, Message, Unit] =
  Pull.output1(line) >> stream.pull.echo
```

The End!

Things we didn’t cover:

- Brackets and safe resource management
- Handling exceptions
- Simpler ways of statefully transforming streams
- Chunks!
- Dealing with multiple execution contexts
- Networking!

If you had fun, come work at Evolution!

We are hiring! Including in Portugal!

Scala & JS/TS engineers!

Visit evolution.com/careers

Or come talk to me and my colleagues at our booth!