Writing applications in Elm
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

cphbusiness

Spring 2017

Outline

Success

Failure
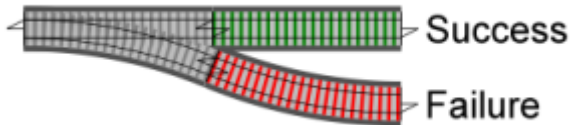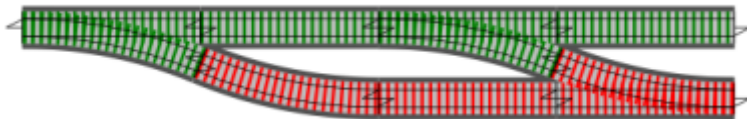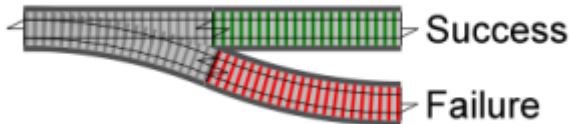
Pipeline: A sequence of functions chained together.

Pipeline: A sequence of functions chained together. **Pipe**: Uses the output of one function as the input to another.

```
(|>) : a -> (a -> b) -> b
```

s

Pipeline: A sequence of functions chained together. **Pipe**: Uses the output of one function as the input to another.

```
(|>) : a -> (a -> b) -> b
```

s

```
function 1 |> function 2
```

Pipeline: A sequence of functions chained together. **Pipe**: Uses the output of one function as the input to another.

```
(|>) : a -> (a -> b) -> b
```

s

```
function 1 |> function 2
```

```
function 1 |> function 2 |> ... |> function n
```

```
["Daniel", "Dennett"]
  |> String.join ""
  |> String.length -- 13
```

JSON parsing in Elm

☐ What is the input?

- What is the input?
    `String`

- ☐ What is the input?
    `String`
- ☐ What is the output?

☐ What is the input?
    `String`

☐ What is the output?
    `Result Http.Error String`

From the package Decode

From the package `Decode`

A decoder decodes to a type a

From the package Decode

A decoder decodes to a type a

```
Decoder a
```

From the package Decode

A decoder decodes to a type a

```
Decoder a
```

```
Decode.int -- simply decodes JSON int to Elm Int
```

```
decodeString : Decoder a -> String -> Result String a
```

Running Elm decoder

```
decodeString : Decoder a -> String -> Result String a
```

```
decodeString int "4"     == Ok 4
decodeString int "1 + 2" == Err ...
```

```
list : Decoder a -> Decoder (List a)
```

```
list : Decoder a -> Decoder (List a)
```

```
list int -- Decoder (List Int)
```

```
jsonString : String
jsonString = """{ "name": "David Chalmers"}"""
```

```
jsonString : String
jsonString = """{ "name": "David Chalmers"}"""
```

```
field : String -> Decoder a -> Decoder a
```

```
jsonString : String
jsonString = """{ "name": "David Chalmers"}"""
```

```
field : String -> Decoder a -> Decoder a
```

```
at "name" string
```

```
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```elm
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```elm
at : List String -> Decoder a -> Decoder a
```

```
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```
at : List String -> Decoder a -> Decoder a
```

```
at ["result", "name"] string
```

```
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```
at : List String -> Decoder a -> Decoder a
```

```
jsonString : String
jsonString =
  """{ "result": { "name": "David Chalmers"}}"""
```

```
at : List String -> Decoder a -> Decoder a
```

```
at ["result", "name"] string
```

```
jsonString : List String
jsonString = """[ "Donald", "Davidson" ]"""
```

```
jsonString : List String
jsonString = """[ "Donald", "Davidson" ]"""
```

```
index : Int -> Decoder a -> Decoder a
```

```
jsonString : List String
jsonString = """[ "Donald", "Davidson" ]"""
```

```
index : Int -> Decoder a -> Decoder a
```

```
index 0 string
```

```
jsonString : String
jsonString = """{ "name": "Richard Dawkins" }"""
```

```
jsonString : String
jsonString = """{ "name": "Richard Dawkins" }"""
```

```
type alias Person
 = { name: String }
```

```
jsonString : String
jsonString = """{ "name": "Richard Dawkins" }"""
```

```
type alias Person
 = { name: String }
```

```
personParser : Decoder Person
personParser =
    field "name" string
```

JSON parsing example

```elm
jsonString : String
jsonString = """{ "name": "Richard Dawkins" }"""
```

```elm
type alias Person
 = { name: String }
```

```elm
personParser : Decoder Person
personParser =
   field "name" string
```

```elm
parsePerson : String -> Result String Person
parsePerson =
   decodeString personParser
```

NoRedInk/elm-decode-pipeline

A library for building decoders using the pipeline (|>) operator and plain function calls.

NoRedInk/elm-decode-pipeline

A library for building decoders using the pipeline (|>) operator and plain function calls.

Install with `elm-package install NoRedInk/elm-decode-pipeline`

Building a pipeline:

Building a pipeline:

```
decode : a -> Decoder a
```

Building a pipeline:

```
decode : a -> Decoder a
```

```
(|>) : a -> (a -> b) -> b
```

Building a pipeline:

```
decode : a -> Decoder a
```

```
(|>) : a -> (a -> b) -> b
```

Required fields:

Building a pipeline:

```
decode : a -> Decoder a
```

```
(|>) : a -> (a -> b) -> b
```

Required fields:

```
required : String -> Decoder a
        -> Decoder (a -> b) -> Decoder b
```

Building a pipeline:

```
decode : a -> Decoder a
```

```
(|>) : a -> (a -> b) -> b
```

Required fields:

```
required : String -> Decoder a
         -> Decoder (a -> b) -> Decoder b
```

Optional fields:

Building a pipeline:

```
decode : a -> Decoder a
```

```
(|>) : a -> (a -> b) -> b
```

Required fields:

```
required : String -> Decoder a
         -> Decoder (a -> b) -> Decoder b
```

Optional fields:

```
optional : String -> Decoder a
         -> a -> Decoder (a -> b) -> Decoder b
```

```
import Json.Decode.Pipeline exposing (..)
```

```
import Json.Decode.Pipeline exposing (..)
```

```
type alias User
  = { id : Int }
```

```elm
import Json.Decode.Pipeline exposing (..)
```

```elm
type alias User
  = { id : Int }
```

```elm
userDecoder : Decoder User
userDecoder =
  decode User
    |> required "id" int
```

```
import Json.Decode.Pipeline exposing (..)
```

```
type alias User
  = { id : Int
    , name : String
    , email : String
    }
```

```
import Json.Decode.Pipeline exposing (..)
```

```
type alias User
  = { id : Int
    , name : String
    , email : String
    }
```

```
userDecoder : Decoder User
userDecoder =
  decode User
    |> required "id" int
    |> required "name" string
    |> optional "email" string "no email"
```

Clone the `elm-exercises` from
`cphbus-functional-programming`

`https://github.com/cphbus-functional-programming/`
`elm-exercises`

Work on the `json.elm` file in the `basicelm` folder

Goal 1: Parse the incoming JSON to a `Person`

Goal 2: Display all the fields in the `Person` type in the HTML

Handled by the `Time` package

Handled by the Time package

```
type alias Time = Float
```

Time can be converted into

☐ Hours: `inHours :  Time -> Float`

Time can be converted into

- ☐ Hours: inHours :   Time -> Float
- ☐ Minutes: inMinutes :   Time -> Float

Time can be converted into

- ☐ Hours: `inHours :  Time -> Float`
- ☐ Minutes: `inMinutes :  Time -> Float`
- ☐ ... and seconds and milliseconds

Periodic updates is something in a fixed interval

Periodic updates is something in a fixed interval

Like `setInterval` in JavaScript

Periodic updates is something in a fixed interval

Like `setInterval` in JavaScript

```
every : Time -> (Time -> msg) -> Sub msg
```

Periodic updates is something in a fixed interval

Like setInterval in JavaScript

```
every : Time -> (Time -> msg) -> Sub msg
```

msg is used in update
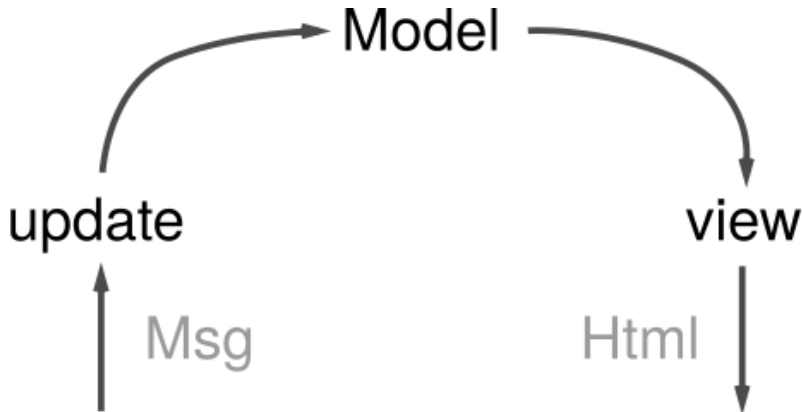
Periodic updates is something in a fixed interval

Like setInterval in JavaScript

```
every : Time -> (Time -> msg) -> Sub msg
```

msg is used in update

```
update : msg -> model -> model
```

```
type Sub msg
```

```
type Sub msg
```

A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"

```
type Sub msg
```

A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"

What's the Input?

```
type Sub msg
```

A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"

What's the Input? And output?

```
type Sub msg
```

> *A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"*

What's the Input? And output?

```
type Msg = Tick Time
```

```
type Sub msg
```

A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"

What's the Input? And output?

```
type Msg = Tick Time
```

```
every : Time -> (Time -> msg) -> Sub msg
```

```
type Sub msg
```

A subscription is a way of telling Elm, "Hey, let me know if anything interesting happens over there!"

What's the Input? And output?

```
type Msg = Tick Time
```

```
every : Time -> (Time -> msg) -> Sub msg
```

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every millisecond Tick
```

Clone the `elm-exercises` from
`cphbus-functional-programming`

`https://github.com/cphbus-functional-programming/`
`elm-exercises`

Work on the `subscriptions.elm` file in the `basicelm` folder

Goal 1: Start a subscription every millisecond

Goal 2: Update the model when the subscription arrive in the update functions

Goal 3: Set the width of the second progress-bar in the view (by correctly updating the 'progress' variable in line 51) to go from 0 to 100 once every 5 seconds.

Defined with the URI prefix: `ws` and `wss` (secure).

Defined with the URI prefix: `ws` and `wss` (secure).

Full duplex between over a single `TCP` channel.

Defined with the URI prefix: `ws` and `wss` (secure).

Full duplex between over a single `TCP` channel.

What are the pros and cons of using websockets?

```
listen : String -> (String -> msg) -> Sub msg
```

```
listen : String -> (String -> msg) -> Sub msg
```

```
type Msg = Echo String | ...

subscriptions model =
  listen "ws://echo.websocket.org" Echo
```

Clone the `elm-exercises` from
`cphbus-functional-programming`

https://github.com/cphbus-functional-programming/
elm-exercises

Work on the `chat.elm` file in the `chat` folder

Goal 1: Start the node websocket server available in `server.js`

Goal 2: Connect to the websocket server on port 3000 from Elm

Goal 3: Write the response from the server to your HTML

Use the following JSON protocol
{ command: "login|send", content: "userName|message" }
and extend the `chat.elm` page to:

1. Send and receive messages in JSON

2. Display a full list of the last chat messages
   This includes changing the model to contain a list of chat messages

3. Login with a user name
   Implement this either by prefixing the user name in the button with "login" (so to log in with "Anders" you write "loginAnders"), or by creating a new input field

Use the following JSON protocol
{ command: "login|send", content: "userName|message" }
and extend the `chat.elm` page to:

1. Send and receive messages in JSON
2. Display a full list of the last chat messages
   This includes changing the model to contain a list of chat messages
3. Login with a user name
   Implement this either by prefixing the user name in the button with "login" (so to log in with "Anders" you write "loginAnders"), or by creating a new input field

Hand in before **17th April 23:59**