Writing applications in Elm
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2017

Outline

HTTP recap

Railroad oriented programming
- Pipelines
- Exercise 1

Debugging

Parsing JSON in Elm
- Decoder pipeline
- Exercise 2

Subscriptions
- Exercise 3

cphbusiness
COPENHAGEN BUSINESS ACADEMY

HTTP in Elm recap

Result

Result

```
type Result error value
  = Ok value
  | Err error
```

Result

```
type Result error value
  = Ok value
  | Err error
```

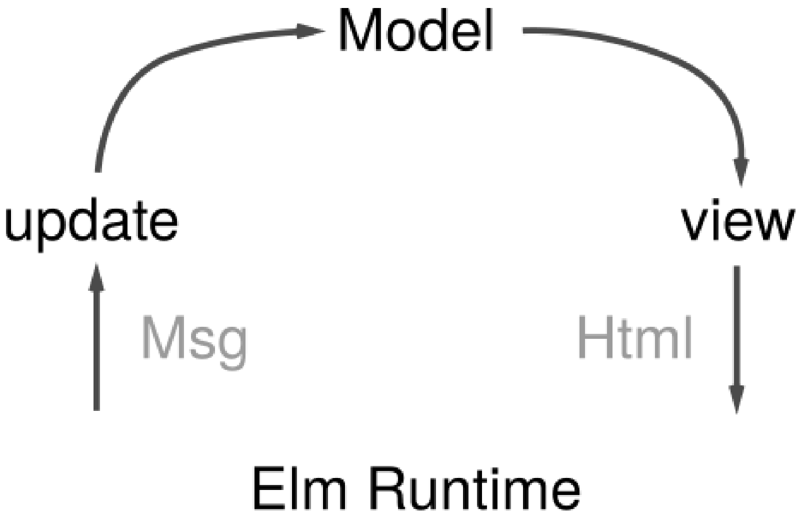
Union type

HTTP request

```
type alias Request a
  = Request a
```

```
getString : String -> Request String
```

HTML updates



HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

```
type Msg
  = NewContent ?
```

HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

```
type Msg
  = NewContent ?
```

```
type Msg
  = NewContent (Result Http.Error String)
```

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

```
HTTP.send : (Result Error a -> msg) ->
         Request a -> Cmd msg
```

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

```
HTTP.send : (Result Error a -> msg) ->
            Request a -> Cmd msg
```

Translated:

- `HTTP.send` takes two parameters

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

```
HTTP.send : (Result Error a -> msg) ->
            Request a -> Cmd msg
```

Translated:

- `HTTP.send` takes two parameters
- 1: One function which takes a result and converts it into something else

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

```
HTTP.send : (Result Error a -> msg) ->
            Request a -> Cmd msg
```

Translated:

- `HTTP.send` takes two parameters
- 1: One function which takes a result and converts it into something else
- 2: One request which performs the HTTP call

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

```
HTTP.send : (Result Error a -> msg) ->
            Request a -> Cmd msg
```

Translated:

- `HTTP.send` takes two parameters
- 1: One function which takes a result and converts it into something else
- 2: One request which performs the HTTP call
- `HTTP.send` returns the message extracted from the first function

Putting it together

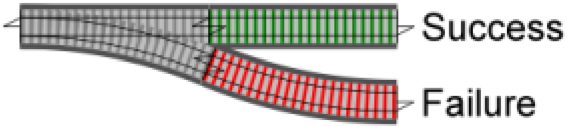
```elm
import Http

type Msg = Click | NewBook (Result Http.Error String)

update : Msg -> Model -> Model
update msg model =
  case msg of
    Click -> ( model, getWarAndPeace )

    NewBook (Ok book) -> ...

    NewBook (Err _) -> ...

getWarAndPeace : Cmd Msg
getWarAndPeace =
  Http.send NewBook <|
    Http.getString "https://example.com/some_book.md"
```

Working with `Result`

Either can have two results: `Ok` and `Err`.

Working with Result

Either can have two results: `Ok` and `Err`.



HTTP results

Working with `HTTP requests`:

- You send something in

HTTP results

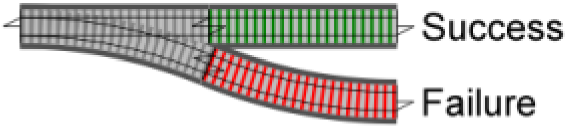
Working with `HTTP requests`:

- You send something in
- You get either a win or a fail

HTTP results

Working with `HTTP requests`:

- You send something in
- You get either a win or a fail



HTTP responses

Working with `HTTP responses`:

- You send something in
- You get either a win or a fail

HTTP responses

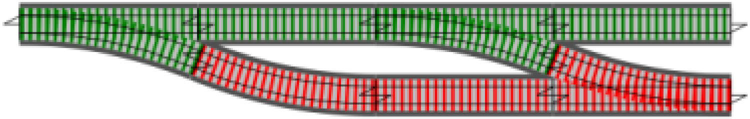
Working with `HTTP responses`:

- You send something in
- You get either a win or a fail
- Then you start parsing!

HTTP responses

Working with `HTTP responses`:

- You send something in
- You get either a win or a fail
- Then you start parsing!



Railroad oriented programming

Cool because:

Railroad oriented programming

Cool because:

- You treat everything in your type system

Railroad oriented programming

Cool because:

- You treat everything in your type system
- You do not run unnecessary code - exit on error

Railroad oriented programming

Cool because:

- You treat everything in your type system
- You do not run unnecessary code - exit on error
- You can piece together modules according to your need

Railroads in Elm

Railroads in Elm

```
getResponseFromUrl
    : String -> Result Http.Error String
```

Railroads in Elm

```
getResponseFromUrl
    : String -> Result Http.Error String
```

```
parseResponseToInt
    : Result Http.Error String
        -> Result Http.Error Int
```

Railroads in Elm

```
getResponseFromUrl
    : String -> Result Http.Error String
```

```
parseResponseToInt
    : Result Http.Error String
        -> Result Http.Error Int
```

```
parseIntToPerson
    : Result Http.Error Int
        -> Result String Person
```

Pipes in Elm 1/2

Pipeline: A sequence of functions chained together.

Pipes in Elm 1/2

Pipeline: A sequence of functions chained together.

Pipe: Uses the output of one function as the input to another.

Pipes in Elm 1/2

Pipeline: A sequence of functions chained together.

Pipe: Uses the output of one function as the input to another.

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

Pipes in Elm 1/2

Pipeline: A sequence of functions chained together.

Pipe: Uses the output of one function as the input to another.

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

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

Pipes in Elm 2/2

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

Pipes in Elm 2/2

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

Forward function application $x \mid> f == f\ x$. This function is useful for avoiding parentheses and writing code in a more natural way.

Pipes in Elm 2/2

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

Forward function application x |> f == f x. This function is useful for avoiding parentheses and writing code in a more natural way.

```
join : String -> List String -> String
```

Pipes in Elm 2/2

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

Forward function application x |> f == f x. This function is useful for avoiding parentheses and writing code in a more natural way.

```
join : String -> List String -> String
```

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

Exercise 1: Implement a railroad

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

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

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

Goal: Count the size of a list of strings by going from *Maybe (List String)* to `Maybe Int`!

Debugging in Elm

```
log : String -> a -> a
```

Debugging in Elm

```
log : String -> a -> a
```

```
toString : Int -> String
toString number
  = Debug.log "Input␣is:␣" (toString number)
```

Crashing Elm

```
crash : String -> a
```

Crashing Elm

```
crash : String -> a
```

Crash the program with an error message. This is an uncatchable error, intended for code that is soon-to-be-implemented."

Holes

Crashing is useful for

- Paying less up-front - partial appliations

Holes

Crashing is useful for

- Paying less up-front - partial appliations
- Verifying control logic

Holes

Crashing is useful for

- Paying less up-front - partial appliations
- Verifying control logic
- Same as holes in Idris

JSON parsing in Elm

JSON parsing in Elm

JSON parsing

- What is the input?

JSON parsing

- What is the input?

  `String`

JSON parsing

- What is the input?

  `String`
- What is the output?

JSON parsing

- What is the input?

  `String`

- What is the output?

  `Result Http.Error String`

JSON decoders

From the package `Decode`

JSON decoders

From the package `Decode`

A decoder decodes to a type `a`

JSON decoders

From the package `Decode`

A decoder decodes to a type a

```
Decoder a
```

JSON decoders

From the package Decode

A decoder decodes to a type a

```
Decoder a
```

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

Running Elm decoder

```
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 ...
```

Combining decoders

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

Combining decoders

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

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

Decoding object fields

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

Decoding object fields

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

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

Decoding object fields

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

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

```
at "name" string
```

Decoding nested objects

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

Decoding nested objects

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

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

Decoding nested objects

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

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

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

Decoding nested objects

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

Decoding nested objects

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

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

Decoding nested objects

cphbusiness COPENHAGEN BUSINESS ACADEMY

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

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

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

Decoding arrays

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

Decoding arrays

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

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

Decoding arrays

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

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

```
index 0 string
```

JSON parsing example

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

JSON parsing example

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

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

JSON parsing example

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

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

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

JSON parsing example

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

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

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

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

Decoder pipeline

NoRedInk/elm-decode-pipeline

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

Decoder pipeline

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:

Building a pipeline

Building a pipeline:

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

Building a pipeline

Building a pipeline:

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

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

Building a pipeline

Building a pipeline:

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

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

Required fields:

Building a pipeline

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

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

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
```

Decoder pipeline example 1/2

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

Decoder pipeline example 1/2

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

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

Decoder pipeline example 1/2

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

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

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

Decoder pipeline example 2/2

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

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

Decoder pipeline example 2/2

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

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

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

Exercise 2: Parse JSON

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

Time in Elm

Handled by the `Time` package

Time in Elm

Handled by the `Time` package

```
type alias Time = Float
```

Converting time in Elm

Time can be converted into

- Hours: `inHours : Time -> Float`

Converting time in Elm

Time can be converted into

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

Converting time in Elm

Time can be converted into

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

Periodic updates

Periodic updates is something in a fixed interval

Periodic updates

Periodic updates is something in a fixed interval

Like `setInterval` in JavaScript

Periodic updates

Periodic updates is something in a fixed interval

Like `setInterval` in JavaScript

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

Periodic updates

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

Periodic updates is something in a fixed interval

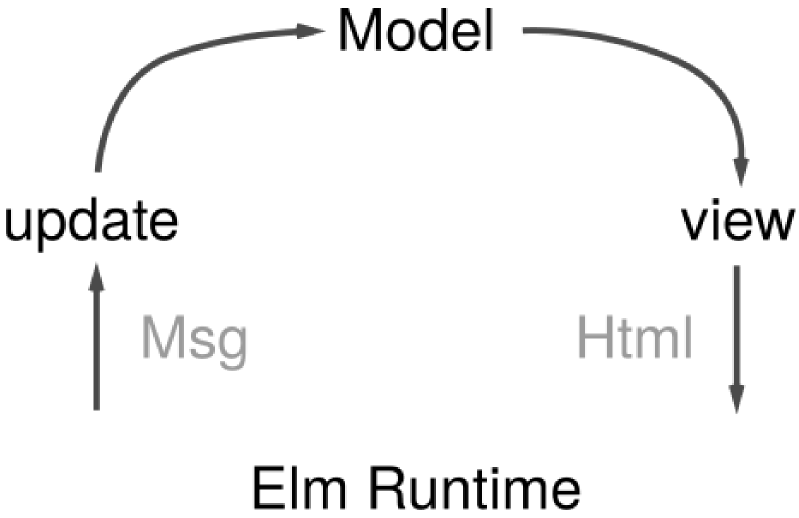
Like `setInterval` in JavaScript

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

`msg` is used in `update`

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

HTML updates



Subscriptions in Elm

```
type Sub msg
```

Subscriptions in Elm

```
type Sub msg
```

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

Subscriptions in Elm

```
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?

Subscriptions in Elm

```
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?

Subscriptions in Elm

```
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
```

Subscriptions in Elm

```
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 in Elm

```
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
```

Exercise 3: Timed progress bar

cphbusiness
COPENHAGEN BUSINESS ACADEMY

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.