Elm - functional programming language for the web

Przemysław Chmielewski, Software Developer > scalac

Who am I?

Which front end framework to choose?

Which front end framework to choose?

Which front end framework to choose?

Which front end framework to choose?

Which front end framework to choose?

Which front end framework to choose?



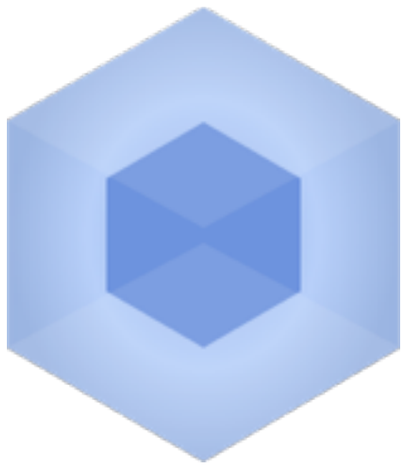
Which front end framework to choose?

Which front end framework to choose?

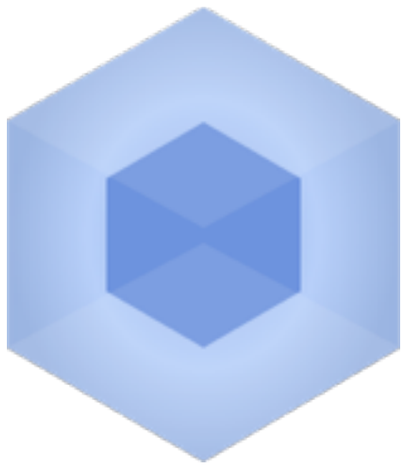
Which front end framework to choose?

Which front end framework to choose?

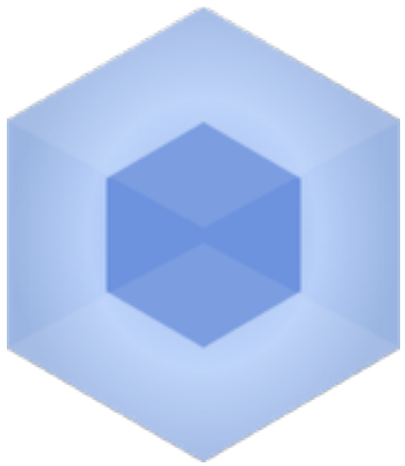
Which front end framework to choose?

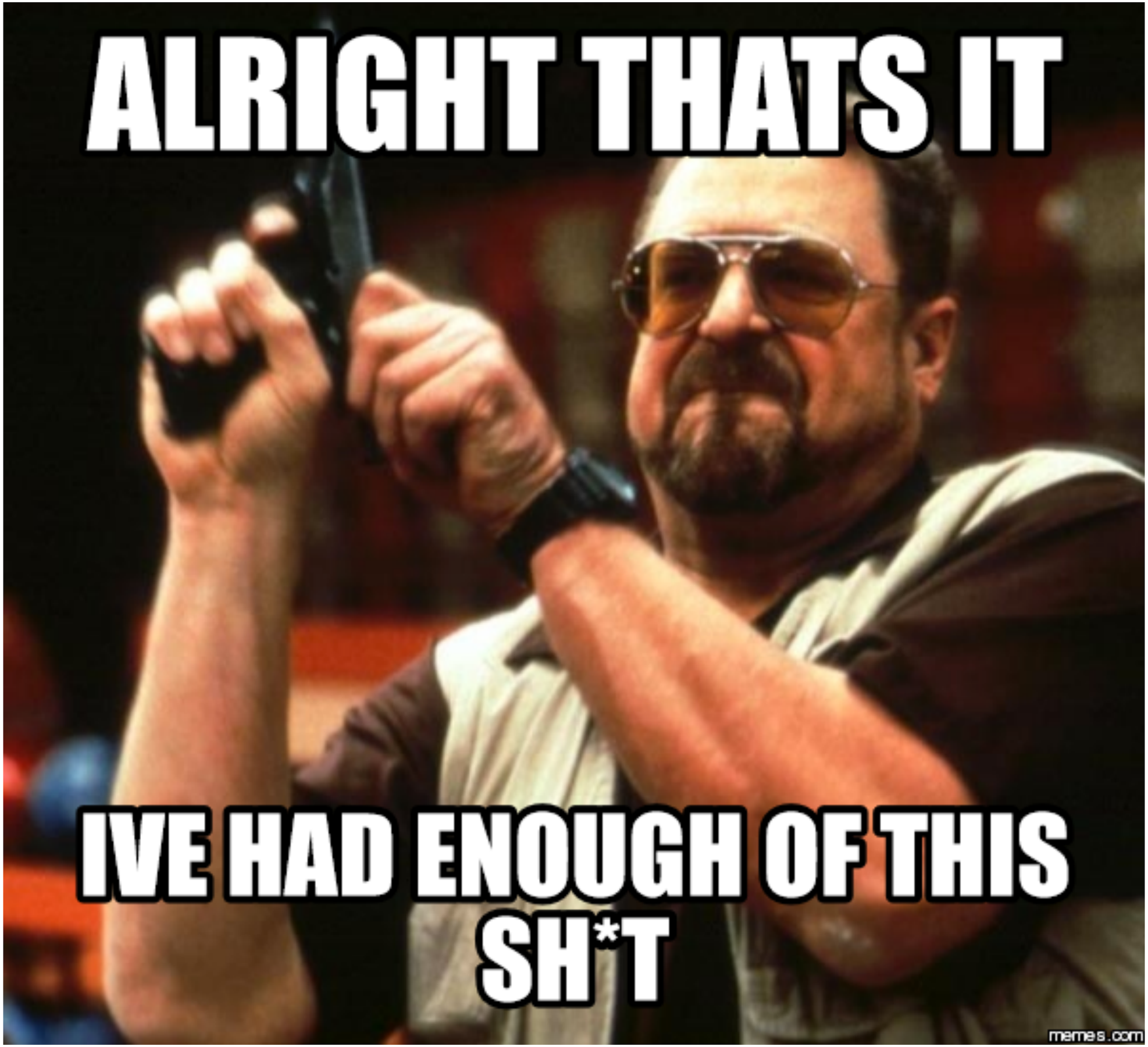


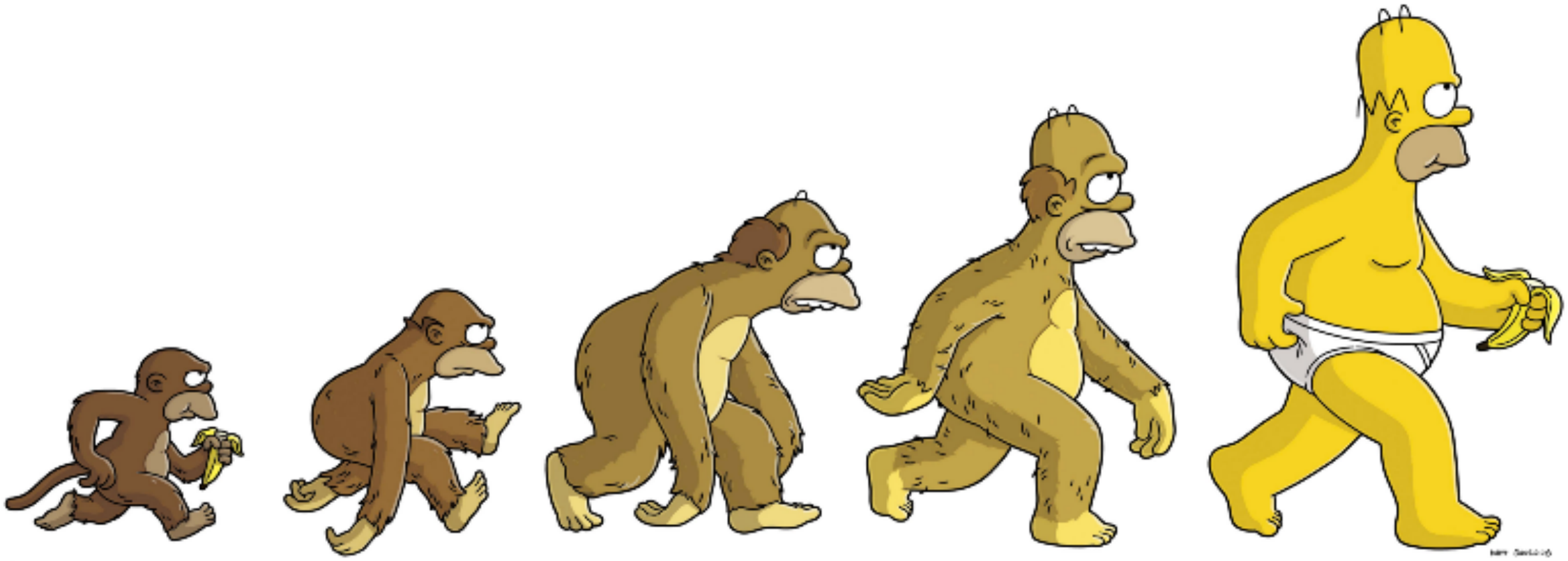
Which front end framework to choose?



Which front end framework to choose?




ALRIGHT THATS IT
IVE HAD ENOUGH OF THIS SH*T
memes.com



MACHINE ASSEMBLY PROCEDURAL OBJECT ORIENTED FUNCTIONAL

Functional programming languages (web)

Functional programming languages (web)

elm

Why Elm?

No **runtime** errors.

No **null**.

No **undefined** is not a function.

Friendly error messages


What did you say?

Friendly error messages

```
1
2  import Html exposing (..)
3
4
5  view users =
6    div [] (List.nap viewUser users)
7
8
9  viewUser user =
10   span [] [text user.name]
11
```

```
-- NAMING ERROR ------------------------------------- naming/unknown-name.elm

Cannot find variable `List.nap`.

6|   div [] (List.nap viewUser users)
             ^^^^^^^^

`List` does not expose `nap`. Maybe you want one of the following?

    List.map
    List.any
    List.map2
    List.map3
```

Friendly error messages

```
1
2   import Html exposing (..)
3   import Html.Attributes exposing (..)
4
5
6   alice =
7     img [src "/users/alice/pic"] []
8
9
10  bob =
11    img [src "/users/bob/pic"] []
12
13
14  userPics =
15    [ alice, bob, "/users/chuck/pic" ]
16
```

```
-- TYPE MISMATCH ------------------------------------------------ types/list.elm

The 3rd element of this list is an unexpected type of value.

15|   [ alice, bob, "/users/chuck/pic" ]
                    ^^^^^^^^^^^^^^^^^^^

All elements should be the same type of value so that we can iterate over the
list without running into unexpected values.

As I infer the type of values flowing through your program, I see a conflict
between these two types:

    Html

    String
```

Friendly error messages

```
1
2  hermann =
3      { first = "Hermann"
4      , last = "Hesse"
5      }
6
7
8  isOver50 person =
9      person.age > 50
10
11
12 answer =
13     isOver50 hermann
14
15
```

```
-- TYPE MISMATCH ------------------------------------ types/missing-field.elm

The 1st argument to function `isOver50` has an unexpected type.

13|    isOver50 hermann
                ^^^^^^^

Looks like a record is missing the field `age`

As I infer the type of values flowing through your program, I see a conflict
between these two types:

    { a | age : comparable }

    { first : String, last : String }
```

Automatically enforced semantic versioning

Good architecture

Elm

1. Core Language
2. Elm Architecture
3. Elm Ecosystem

Elm language - literals

```
True  : Bool
False : Bool

42    : number  -- Int or Float depending on usage
3.14  : Float
```

```
'a'   : Char
"abc" : String
```

```
[1, 2, 3, 4, 5]                  : List number
1 :: [2, 3, 4, 5]                : List number
1 :: 2 :: 3 :: 4 :: 5 :: []      : List number

[1.0, 2.0, 3.0, 4.0, 5.0]        : List float
```

Elm language - Conditionals

```
if key == 40 then
    n + 1
else if key == 38 then
    n - 1
else
    n
```

Elm language - Conditionals

```
if key == 40 then
    n + 1
else if key == 38 then
    n - 1
else
    n
```

```
if powerLevel > 9000 then "OVER 9000!!!" else "meh"
```

Elm language - Conditionals

```
if key == 40 then
    n + 1
else if key == 38 then
    n - 1
else
    n
```

```
if powerLevel > 9000 then "OVER 9000!!!" else "meh"
```

```
case maybe of
  Just xs -> xs
  Nothing -> []

case xs of
  hd::tl -> Just (hd,tl)
  []     -> Nothing
```

Elm language - Union Types

```
type Some
    = New
    | InProgress
    | Done
    | Error String

status = New
```

```
showStatus status =
    case status of
        New -> "new"
        InProgress -> "progress"
        Done -> "done"
        Error msg -> "error: " ++ msg

showStatus New                  -- "new"
showStatus (Error "WTF?")       -- "error: WTF?"
```

Elm language - records

```
bill =
  { name = "Gates", age = 57 }

steve =
  { name = "Jobs", age = 56 }

people =
  [ bill, steve]

point2D =
  { x = 0, y = 0}

point3D =
  { x = 3, y = 4, z = 12}
```

Elm language - records

```elm
bill =
  { name = "Gates", age = 57 }

steve =
  { name = "Jobs", age = 56 }

people =
  [ bill, steve]

point2D =
  { x = 0, y = 0}

point3D =
  { x = 3, y = 4, z = 12}
```

```elm
point3D.z                  -- 12
bill.name                  -- "Gates"
.name bill                 -- "Gates"
List.map .age people       -- [57,56]

.x point2D      -- 0
.x point3D      -- 3
.x { x = 4 }    -- 4
```

Elm language - records

```
bill =
  { name = "Gates", age = 57 }

steve =
  { name = "Jobs", age = 56 }

people =
  [ bill, steve]

point2D =
  { x = 0, y = 0}

point3D =
  { x = 3, y = 4, z = 12}
```

```
point3D.z                  -- 12
bill.name                  -- "Gates"
.name bill                 -- "Gates"
List.map .age people       -- [57,56]

.x point2D       -- 0
.x point3D       -- 3
.x { x = 4 }     -- 4

under50 {age} =
  age < 50
```

Elm language - records

```
bill =
  { name = "Gates", age = 57 }

steve =
  { name = "Jobs", age = 56 }

people =
  [ bill, steve]

point2D =
  { x = 0, y = 0}

point3D =
  { x = 3, y = 4, z = 12}
```

```
point3D.z                    -- 12
bill.name                    -- "Gates"
.name bill                   -- "Gates"
List.map .age people         -- [57,56]

.x point2D       -- 0
.x point3D       -- 3
.x { x = 4 }     -- 4

under50 {age} =
  age < 50
```

```
  { point2D | y = 1 }              -- { x=0, y=1 }
  { point3D | x = 0, y = 0 }       -- { x=0, y=0, z=12 }
  { steve | name = "Wozniak" }     -- { name="Wozniak", age=56 }
```

Elm language - type annotations

```elm
answer : Int
answer =
  42

factorial : Int -> Int
factorial n =
  List.product [1..n]

distance : { x : Float, y : Float } -> Float
distance {x,y} =
  sqrt (x^2 + y^2)
```

Elm language - type aliases

```elm
origin : { x : Float, y : Float }
origin = { x = 0, y = 0 }

type alias Point =
  { x : Float, y : Float }

origin : Point
origin = { x = 0, y = 0}

type alias Positioned a =
  { a | x : Float, y : Float }

type alias Named a =
  { a | name : String }

type alias Moving a =
  { a | velocity : Float, angle : Float }
```

```elm
lady : Named { age:Int }
lady =
  { name = "Lois Lane"
  , age = 31
  }

dude : Named (Moving (Positioned {}))
dude =
  { x = 0
  , y = 0
  , name = "Clark Kent"
  , velocity = 42
  , angle = degrees 30
  }
```

Elm language - functions

```
sayHello name =
    String.append "Hello " name

names = ["Adam", "Przemek", "Andrzej"]

List.map sayHello names      -- ["Hello Adam","Hello Przemek","Hello Andrzej"]
```

Elm language - anonymous functions

```elm
greet name  = "Hello, " ++ name
greet = \name -> "Hello, " ++ name

greet "Adam"                                -- "Hello, Adam"

numbers = List.range 1 10                   -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List.filter (\x -> x < 5) numbers           -- [1, 2, 3, 4]
```

Elm language - piping

```
String.repeat 3 (String.toUpper (String.append "h" "i"))
-- "HIHIHI" : String

String.append "h" "i" |> String.toUpper |> String.repeat 3
-- "HIHIHI" : String

String.repeat 3 <| String.toUpper <| String.append "h" "i"
-- "HIHIHI" : String
```

Elm language - piping

```
String.repeat 3 (String.toUpper (String.append "h" "i"))
-- "HIHIHI" : String

String.append "h" "i" |> String.toUpper |> String.repeat 3
-- "HIHIHI" : String

String.repeat 3 <| String.toUpper <| String.append "h" "i"
-- "HIHIHI" : String



String.append "h" “i"
  |> String.toUpper
  |> String.repeat 3
```

Elm language - currying

```
threeTimes = String.repeat 3
-- <function> : String -> String

threeTimes "hi"
-- "hihihi" : String
```

Elm language - Maybe

```elm
type Maybe a
    = Nothing
    | Just a

type alias User =
  { name : String
  , age : Maybe Int
  }

sue : User
sue =
  { name = "Sue", age = Nothing }

tom : User
tom =
  { name = "Tom", age = Just 24 }
```

```elm
canBuyAlcohol : User -> Bool
canBuyAlcohol user =
  case user.age of
    Nothing ->
      False

    Just age ->
      age >= 21

canBuyAlcohol sue          -- False
canBuyAlcohol tom          -- True
```

Elm language - Result

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

String.toInt x              -- String -> Result.Result String Int
String.toInt "10"           -- Ok 10 : Result.Result String Int
String.toInt "asdas"        -- Err "could not convert string 'asdas' to an Int" :
                               Result.Result String Int

isValidAge : String -> Bool
isValidAge age =
  case String.toInt age of
    Err msg ->
      False

    Ok age ->
      True

isValidAge "10"                 -- True
isValidAge "asdas"              -- False
```

Elm language - Task

```
type alias Task err ok =
    Task err ok

Task.succeed 42                 -- Task x String
Task.fail "reason"              -- Task String a
```

Elm language - Task

```
type alias Task err ok =
    Task err ok

Task.succeed 42                    -- Task x String
Task.fail "reason"                 -- Task String a

type Msg =
    CurrentTime Time

Task.perform : (a -> msg) -> Task Never a -> Cmd msg
Task.perform CurrentTime Time.now

Time.now : Task x Time
```

Elm language - Task

```elm
type alias Task err ok =
    Task err ok

Task.succeed 42                     -- Task x String
Task.fail "reason"                  -- Task String a

type Msg =
    CurrentTime Time
    | CurrentUser (Result Error User)

Task.perform : (a -> msg) -> Task Never a -> Cmd msg
Task.perform CurrentTime Time.now

Time.now : Task x Time

Task.attempt : (Result x a -> msg) -> Task x a -> Cmd msg
Task.attempt CurrentUser GetCurrentUser

GetCurrentUser : Task Error User
```

Elm language - modules

```elm
module MyModule exposing (..)

-- qualified imports
import List                         -- List.map, List.foldl
import List as L                    -- L.map, L.foldl

-- open imports
import List exposing (..)                   -- map, foldl, concat, ...
import List exposing ( map, foldl )         -- map, foldl

import Maybe exposing ( Maybe )             -- Maybe
import Maybe exposing ( Maybe(..) )         -- Maybe, Just, Nothing
import Maybe exposing ( Maybe(Just) )       -- Maybe, Just
```

Elm language - Interop Html

```
<div id="main"></div>
<script src="main.js"></script>
<script>
    var node = document.getElementById('main');
    var app = Elm.Main.embed(node);
</script>
```

```
elm-make src/Main.elm
elm-make src/Main.elm --output=main.js
```

Elm language - Interop Javascript Ports

```
<div id="spelling"></div>
<script src="spelling.js"></script>
<script>
    var app = Elm.Spelling.fullscreen();

    app.ports.check.subscribe(function(word) {
        var suggestions = spellCheck(word);
        app.ports.suggestions.send(suggestions);
    });

    function spellCheck(word) {
        // have a real implementation!
        return [];
    }
</script>
```

```
port module Spelling exposing (..)

...
-- port for sending strings out to JavaScript
port check : String -> Cmd msg

-- port for listening for suggestions from JavaScript
port suggestions : (List String -> msg) -> Sub msg
...
```

Elm language - Interop Javascript Flags

```
var node = document.getElementById('my-app');
var app = Elm.MyApp.embed(node, {
    user: 'Tom',
    token: '12345'
});
```

```
type alias Flags =
  { user : String
  , token : String
  }

init : Flags -> ( Model, Cmd Msg )
init flags =
  ...

main =
  programWithFlags { init = init, ... }
```

Elm architecture

```
import Html exposing (..)

-- MODEL
type alias Model = { ... }


-- UPDATE
type Msg = Reset | ...

update : Msg -> Model -> Model
update msg model =
  case msg of
    Reset -> ...
    ...

-- VIEW
view : Model -> Html Msg
view model =
  ...

main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = \_ -> Sub.none
        }
```

Elm architecture - code example

Elm ecosystem

Elm Package

```
elm-package install elm-lang/html          # Install latest version
elm-package install elm-lang/html 1.0.0    # Install version 1.0.0

elm-package diff elm-lang/core 3.0.0 4.0.0

elm-package publish

elm-package bump
```

Elm Package

elm-package.json

```
{
    "version": "1.0.0",
    "summary": "Project summary",
    "repository": "url",
    "license": "BSD3",
    "source-directories": [
        "."
    ],
    "exposed-modules": [],
    "dependencies": {
        "elm-lang/core": "5.0.0 <= v < 6.0.0",
        "elm-lang/dom": "1.1.1 <= v < 2.0.0",
        "elm-lang/html": "2.0.0 <= v < 3.0.0",
        "elm-lang/http": "1.0.0 <= v < 2.0.0"
    },
    "elm-version": "0.18.0 <= v < 0.19.0"
}
```

Elm Package

- Versions all have exactly three parts: MAJOR.MINOR.PATCH
- All packages start with initial version 1.0.0
- Versions are incremented based on how the API changes:
    - PATCH - the API is the same, no risk of breaking code
    - MINOR - values have been added, existing values are unchanged
    - MAJOR - existing values have been changed or removed
- elm-package will bump versions for you, automatically enforcing these rules

Elm REPL

Elm Reactor

Elm Make

Pros

Cons

Questions?

About