AN EXPLORATION OF ELM

PART 1 - BACKGROUND

PART 2 - INTRO TO ELM

PART 3 - WHY ELM IS DIFFERENT

PART 4 - TYPE SYSTEM

PART 5 - UP AND RUNNING

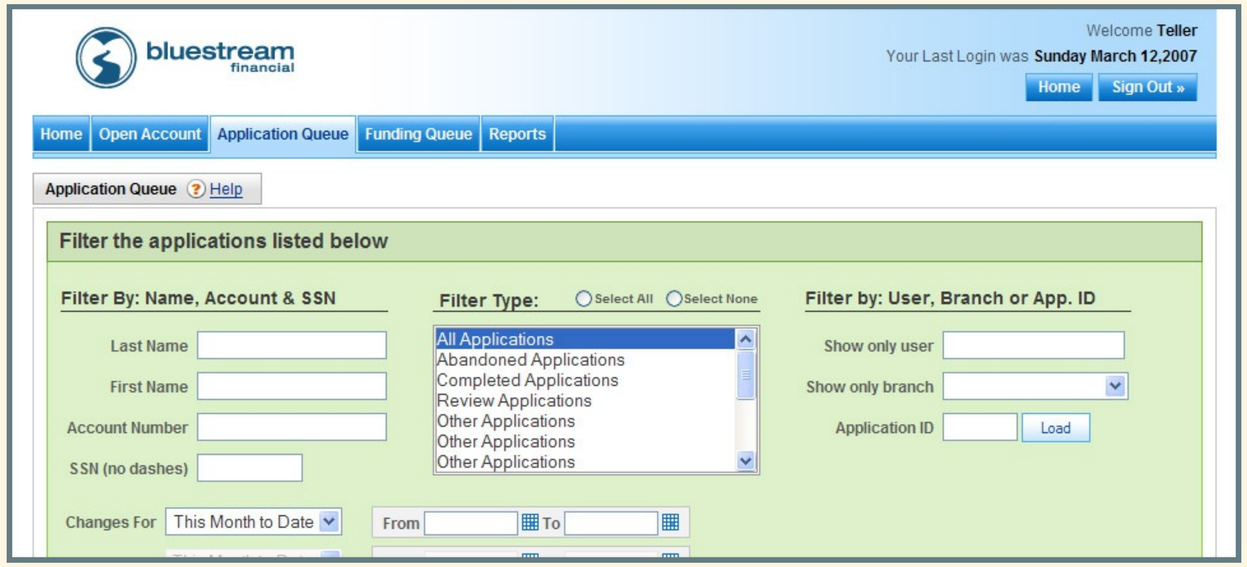
STATE

- All the information pertaining to your program at a given instant in time.

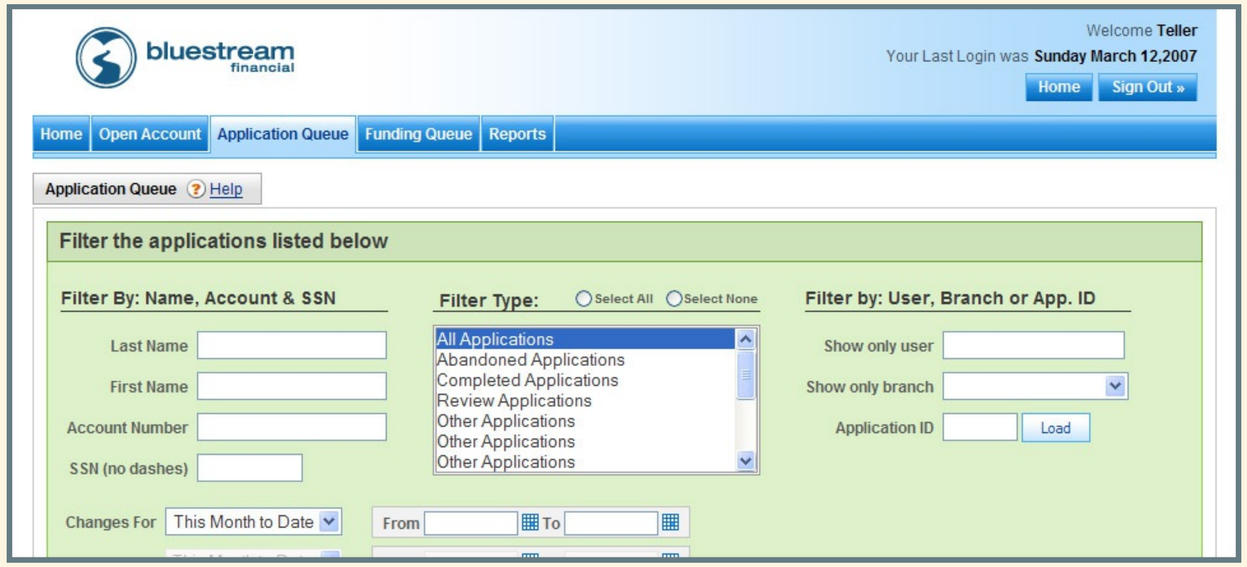
BEHAVIOR

- How state is displayed and changed over time ...*logic*
- Changes made to the external world ...*side effects*

INTERACT WITH A WEBAPP OVER TIME...



WHERE DO DEVELOPERS TRACK ALL THESE CHANGES?



TRADITIONAL ANSWER:

ALL OVER THE PLACE

USE THE TOOLS AVAILABLE TO US

EARLY DAYS...

JAVASCRIPT

DOM

COMMON TO STORE STATE IN THE DOM

```
<table>
    <thead>
        <tr>
            <th id="tHead" data-format-string="Do MMMM YYYY">
        ...
      </th>
            ...
        </tr>
    </thead>
    <tr>
        <td id="dCell" data-value="975683133000"><!-- 1st December 2000
    </tr>
    ...
</table>
```

READ AND WRITE TO THE DOM

```
<table>
    <thead>
        <tr>
            <th id="tHead" data-format-string="Do MMMM YYYY">
        ...
      </th>
            ...
        </tr>
    </thead>
    <tr>
        <td id="dCell" data-value="975683133000"><!-- 1st December 2000
    </tr>
    ...
</table>
```

```
var dateFormat = document.getElementById('tHead');
var dateCell = document.getElementById('dCell');

dateFormat.dataset.formatString // "Do MMMM YYYY"
dateCell.dataset.indexNumber // "975683133000"
// format the date
dateCell.innerHTML = formattedDate;
```

LOTS OF DOM WORK

SO WE INVENTED DOM TOOLS

- jQuery
- Scriptaculous
- Prototype
- Mootools
- etc...

WE'VE ALL BEEN THERE

PRODUCTION PROJECT

HAS AN INCONSISTENCY

YOU'RE BUSY

MULTIPLE PROJECTS
UPCOMING DEADLINES

IT'S ON YOU

- REPRODUCE
- TRACK IT DOWN
- FIX IT

YOU THINK YOU'VE FOUND IT!

APPLY THE FIX...

STILL BROKEN!!!

I'M NOT SUPPOSED TO BE WORKING ON THIS TODAY!

WHY DIDN'T THAT FIX IT?

- Same state is in two different places?
- More than one thing can change this?

MAYBE YOU DID FIX IT?

BUT YOU BROKE SOMETHING ELSE!

DATA IS BEING CHANGED FROM

MULTIPLE PLACES

EVENT HANDLERS FIRE

UNPREDICTABLE ORDER

MULTIPLE PEOPLE HAVE APPLIED "QUICK FIXES"

FEAR

OF BREAKING EXISTING CODE

YOU FINALLY FIX IT

BUT NOTICE SOME WEIRD EDGE-CASE OR BEHAVIOR...

BUT YOU

DON'T HAVE TIME

TO TRACK DOWN THE CAUSE

YOU'RE SUPPOSED TO JUST

FIX THE BUG

NOT REFACTOR AN OLD PROJECT

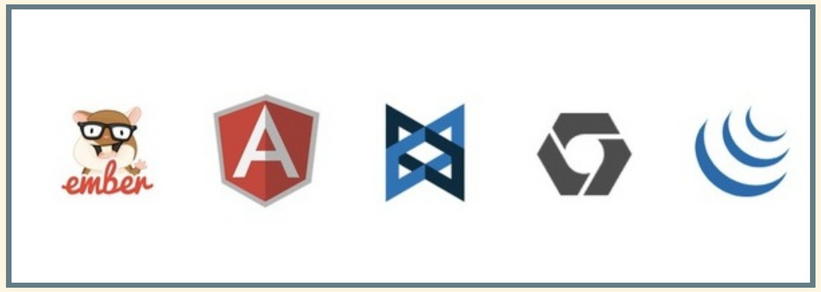
SO YOU WEARILY ADD ANOTHER

IF/ELSE

...AND MOVE ON

AND THAT'S HOW MONSTERS ARE CREATED

MAKE THINGS NICER



- Models
- Templates
- Event bus
- Two way data-binding


BUT, THE PICTURE IS
WONKY

WHEN A CHANGE IS WANTED

- scrap the previous version
- create a new one - almost identical
- but it has this one change

YOU CAN DO THIS WITH THE DOM

DATA

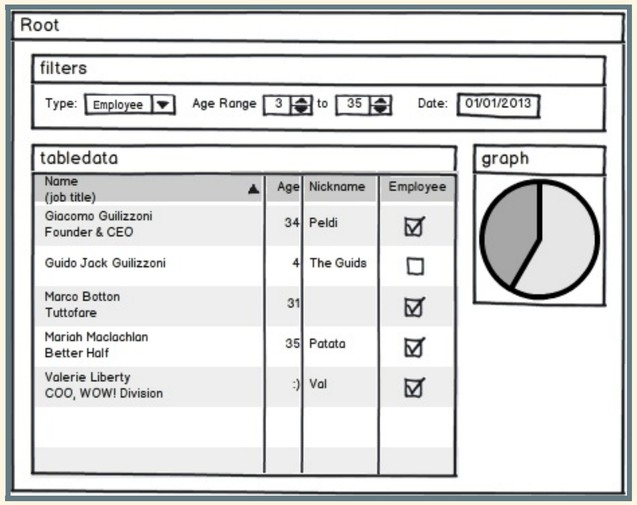
FUNCTION(DATA)

DOM

<COMPONENTS/>

- initialized with properties
- own and manage their own state
- can be nested inside each other
- render DOM elements

WE NEST THE BOXES



RENDER() FUNCTION RETURNS A

DESCRIPTION

...of the DOM

AS DATA

`virtualdom(data)`

...SIDE EFFECT

Virtual DOM is an

EFFECT MANAGER

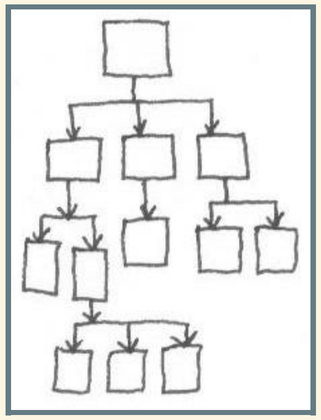
Works on your behalf...

- queue and batch update render "requests"
- minimize DOM mutation
- output cross-platform HTML

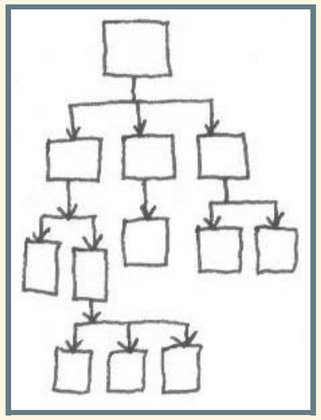
ALMOST FORGET ABOUT THE DOM

DATA IN, DOM OUT

EACH COMPONENT HAS IT'S OWN STATE



STATE IS...



- Still spread out
- Sometimes duplicated

CENTRALIZE THE STATE

PASS IT DOWN

SINGLE
SOURCE OF TRUTH

TO CHANGE THE STATE

COMPONENTS DISPATCH MESSAGES

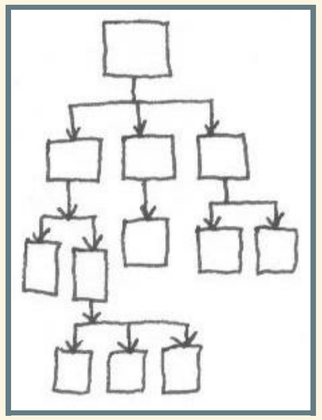
OWNER OF THE STATE

CREATES A NEW STATE

IN RESPONSE

AND PASSES IT

DOWN



Redux

WITH A SINGLE STATE OBJECT ANYONE CAN

MUTATE STATE

WITH OBJECT REFERENCES

SO NOW WE MAKE OUR STATE OBJECT

IMMUTABLE

- Immutable.JS
- SeamlessImmutable
- ...

LOOKING GOOD SO FAR

INVENTORY WAS REFACTORED

```
// It used to be an array...
var inventory = [
  {name: "Gloves", quantity: 3},
  {name: "Valves", quantity: 7}
]

// But now it's an object...
var inventory = {
  assignedTo: "WAREHOUSE 1",
  items: [
    {name: "Gloves", quantity: 3},
    {name: "Valves", quantity: 7}
  ]
}
```

IT BROKE OUR CODE

```
render() {
  if (this.props.inventory.length) {
    return (
      <div>
        this.props.inventory.map(x => {
          return <InventoryItem item={x}/> });
      </div>
    );
  } else {
    return "There are no inventory items.";
  }
}
```

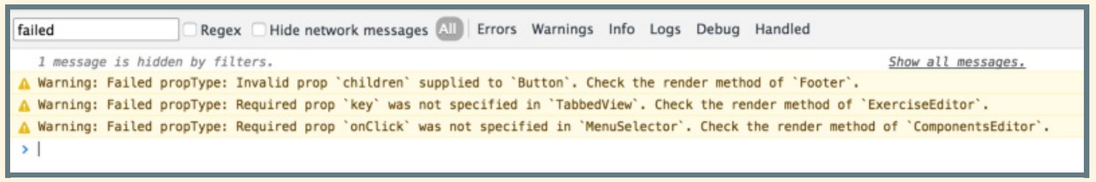
BUT NO ERROR IS THROWN

TOOLS ARE INVENTED

YOU TRY IT, AND GET AN EARLY WARNING

```
propTypes: {
  inventory: React.PropTypes.object.isRequired
}
...
render() {
  if (this.props.inventory.items.length) {
    return (
      <div>
        this.props.inventory.map(x => {
          return <InventoryItem item={x}/>
        });
      </div>
    );
  } else {
    return "There are no inventory items.";
  }
}
```

THIS IS GREAT!



YOU FIX ALL THE WARNINGS

WE'VE COME A LONG WAY!

Early JS	Recent JS
Mutating the DOM manually	Virtual DOM
State was spread out	Now it's centralized
Multi-directional data flow	Uni-directional data flow
Mutable shared state	Immutable shared state
Silent type errors	Type checking UI props

PART 1 - BACKGROUND

PART 2 - INTRO TO ELM

PART 3 - WHY ELM IS DIFFERENT

PART 4 - TYPE SYSTEM

PART 5 - UP AND RUNNING

BABEL

elm

VALUES

```
> "hello"
"hello" : String

> "hello" ++ "world"
"helloworld" : String

> "hello" ++ " world"
"hello world" : String
```

```
> 2 + 3 * 4
14 : Int

> (2 + 3) * 4
20 : Int

> 9 / 2
4.5 : Float

-- Integer division
> 9 // 2
4 : Int
```

IF EXPRESSIONS

```
> if True then
    "hello"
  else
    "world"

"hello" : String

> if False then
    "hello"
  else
    "world"

"world" : String
```

EQUALITY

All of Elm is immutable.

```
> 1 == 2
False : Bool

> "banana" == "banana"
True : Bool
```

Everything's a value *(no ===)*

STRONGLY TYPED

```
> 1 == "1"
-- TYPE MISMATCH ---------------------------------------------- repl-temp

The right argument of (==) is causing a type mismatch.

3|   1 == "1"
          ^^^
(==) is expecting the right argument to be a:

    number

But the right argument is:

    String
```

LISTS

Lists hold values of the same type.

```
> names = [ "Alice", "Bob", "Eve" ]
["Alice","Bob","Eve"]

> List.isEmpty names
False

> List.length names
3

> List.reverse names
["Eve","Bob","Alice"]

> double n = n * 2
<function>

> List.map double [1,2,3,4]
[2,4,6,8]
```

RECORDS

```
> point =
    { x = 3
    , y = 4
    }

> point.x
3

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

> bill.name
"Gates"
```

`.name` is a function that gets the name field of the record.

```
> .name bill
"Gates"

> List.map .name [bill,bill,bill]
["Gates","Gates","Gates"]
```

TUPLES

A tuple holds a **fixed** number of values

```
import String

validateName name =
  if String.length name <= 20 then
    (True, "name accepted!")
  else
    (False, "name was too long; please limit it to 20 characters")

> validateName "Tom"
(True, "name accepted!")
```

Each value can have any type

```
> (12.6, "banana", ["a", "b", "c"], False, {name = "Bill"})

  : ( Float, String, List String, Bool, { name : String } )
```

FUNCTIONS

```
isNegative n =
  n < 0

> isNegative 4
False : Bool

addThese this that =
  this + that

> addThese 7 9
16 : number

-- anonymous functions
squares = List.map (\x -> x * x) [-3,-2,-1,0,1,2,3]
[9,4,1,0,1,4,9] : List number
```

Functions are auto-curried.

```
> divide x y =
    x / y
<function> : Float -> Float -> Float
```

```
> divideTwelve = divide 12
<function> : Float -> Float

> divideTwelve 3
4 : Float
```

Something similar in JavaScript

```
var myCurriableFunction = function (x, y, z) {
  return function (y, z) {
    x + y + z;
  }
}

myCurriableFunction(2)(3, 4)
```

All functions are auto-curried in Elm.

```
String.repeat 3 "hi"
-- "hihihi" : String

String.repeat
-- <function:repeat> : Int -> String -> String

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

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

This allows for elegant composition (pipelining)...

```
"Fort Collins"
  |> String.repeat 2
  |> String.reverse
  |> String.map (\char -> if char == 'o' then 'x' else char)
  |> String.toUpper

"SNILLXC TRXFSNILLXC TRXF" : String
```

BACK TO THE PLURALIZER EXAMPLE

```
pluralize singular plural quantity =
    if quantity == 1 then
        singular
    else
        plural
```

Let's add some interactivity...

TYPES

One of Elm's major benefits is that users do not see runtime errors in practice.

This is because of **type inference**. The compiler figures out what type of values flow in and out of ***all*** your functions.

```
-- TYPE MISMATCH -------------------------------------------------------

The argument to function `toFullName` is causing a mismatch.

6|   toFullName { fistName = "Hermann", lastName = "Hesse" }
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Function `toFullName` is expecting the argument to be:

    { …, firstName : … }

But it is:

    { …, fistName : … }

Hint: I compared the record fields and found some potential typos.

    firstName <-> fistName
```

No matter how big and complex things get, the Elm compiler checks that **everything** fits together properly just based on the source code.

You can actively use Elm's **type system** to improve the **correctness** and **maintainability** of your code.

BACK TO THE PLURALIZER EXAMPLE

```
pluralize singular plural quantity =
    if quantity == 1 then
        singular
    else
        plural
```

Let's introduce some

- type annotations
- a new custom type

PIT OF SUCCESS

The **good practices** that we're seeing convergence on in JS land, are inescapable in Elm.

TODO: maybe kill?

Early JS	Recent JS	React	Elm
Mutating the DOM manually	Virtual DOM	React	Elm
State was spread out	Now it's centralized	Redux	Elm
Multi-directional data flow	Uni-directional data flow	Redux	Elm
Mutable shared state	Immutable shared state	Immutable.js	Elm
Silent type errors	Type checking UI props	PropTypes/Flow	Elm

JS Frameworks are now Emulating Elm

- http://redux.js.org/
- https://github.com/dvajs/dva
- https://github.com/yoshuawuyts/choo

PART 1 - BACKGROUND

PART 2 - INTRO TO ELM

PART 3 - WHY ELM IS DIFFERENT

PART 4 - TYPE SYSTEM

PART 5 - UP AND RUNNING

Every Elm program is composed 100% of

PURE FUNCTIONS

BUT WHAT IS A PURE FUNCTION?

Can anyone spot the hidden **side-effect**?

```
todoList.addTodo = function() {
  todoList.todos.push({text: todoList.todoText, done: false});
  todoList.todoText = '';
}
```

A function that **returns nothing** can only be called for its side effects.

Can anyone spot the **side-cause**?

```
todoList.remaining = function() {
  var count = 0;
  angular.forEach(todoList.todos, function(todo) {
    count += todo.done ? 0 : 1;
  });
  return count; // We ARE returning something this time.
}
```

THERE ARE NO ARGUMENTS

- So either: return the **same value** every time
- There is a **hidden argument**
  - Changes the behavior of this code each time it's called.

EVERY LANGUAGE SUPPORTS PURE FUNCTIONS

```
var adder = function(a, b) {
  return a + b;
}
```

So what makes a language functional?

KRIS JENKINS:

*"Functional Programming is about **eliminating side effects** where you can and **controlling them** where you can't, and that will lead you on a very interesting journey."*

KRIS JENKINS:

"It is my deep hope that in the morning, you will go to your computer, and you will see side-effects everywhere."

KRIS JENKINS:

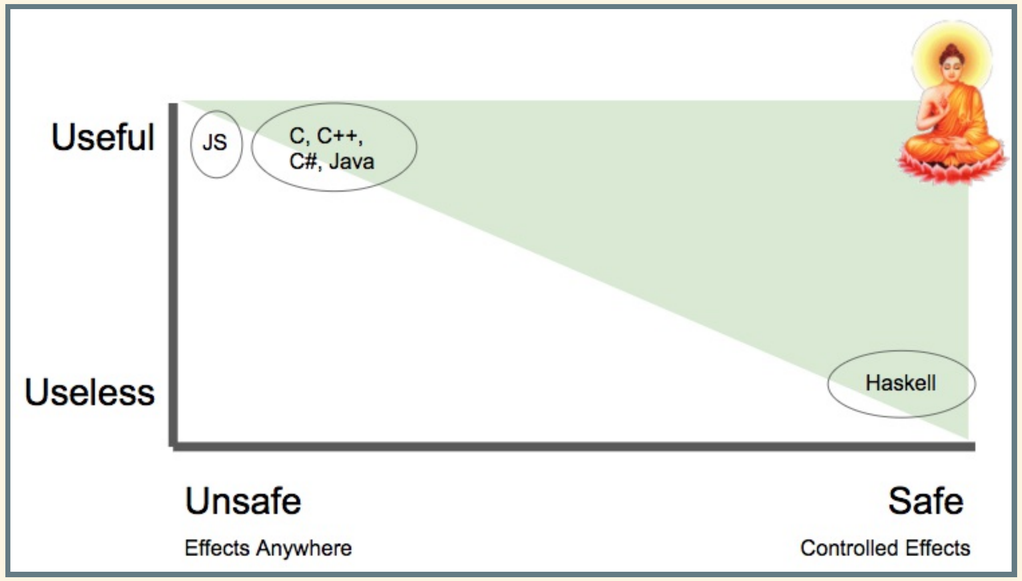
"And it will ruin your lives, and torture you for years..."

KRIS JENKINS:

"...and it will also make you a much better programmer." - @krisajenkins

SIMON PEYTON JONES

- Researcher at Microsoft Research
- https://www.youtube.com/v/iSmkqocn0oQ?start=18&end=115


Useful
Useless
JS
C, C++,
C#, Java
Haskell
Unsafe
Effects Anywhere
Safe
Controlled Effects

Every Elm program is composed 100% of

PURE FUNCTIONS

(no side-effects)

So, how do we do anything **useful**?

Remember the Virtual DOM?

```
render() {
  if (this.props.inventory.length) {
    return (
      <div>
        this.props.inventory.map(x => {
          return <InventoryItem item={x}/> });
      </div>
    );
  } else {
    return "There are no inventory items.";
  }
}
```

`render()` function returns a

DESCRIPTION

...of the DOM **as data**

`virtualdom(data)`

...SIDE EFFECT

Virtual DOM is an

EFFECT MANAGER

...for the DOM

Elm is an

EFFECT MANAGER

...for **EVERYTHING**

TIME

TIME

RANDOMNESS

TIME

RANDOMNESS

DOM

TIME

RANDOMNESS

DOM

HTTP

TIME

RANDOMNESS

DOM

HTTP

WEBSOCKETS

TIME

RANDOMNESS

DOM

HTTP

WEBSOCKETS

STORAGE

TIME

RANDOMNESS

DOM

HTTP

WEBSOCKETS

STORAGE

...you get the idea.

Pure functions describe these effects

AS DATA

Something produces a `Cmd`...

...Elm returns a `Msg`

COMMANDS CAN COME FROM

- `Html` - the user, event in the HTML
- `Cmd` - your code, returning `Cmd` from a function
- `Sub` - a subscription, time, websocket etc

You will **never** see code like this in Elm...

Time

```
var timeInMs = Date.now()
//-> 1477950154376

timeInMs = Date.now() // no arguments
timeInMs = Date.now() // each call returns a different value
```

Randomness

```
var randomFloat = Math.random()
//-> 0.4274841726102241

randomFloat = Math.random() // no arguments
randomFloat = Math.random() // each call returns a different value
```

You ask for these things with a `Cmd`...

...Elm returns them to you in a `Msg`

TRADEOFF

We either move **simple** things (time, random) to **controlled** environment

Or we move **complex** things (your entire codebase) to **uncontrolled** environment

This is the **major difference** between Elm and its competitors...

- All functions are pure
- All side effects are controlled

That means asynchronous and side-effectful functions are:

- easily understood (and usually smaller)
- easily identified by their return type: `Cmd`
- easily testable (data in, data out, no mocking)

This is a huge benefit for **maintanability**.

PART 1 - BACKGROUND

PART 2 - INTRO TO ELM

PART 3 - WHY ELM IS DIFFERENT

PART 4 - TYPE SYSTEM

PART 5 - UP AND RUNNING

Elm **lifts** side effects into its type system.

`Cmd`

But there are many other hazards for our programs...

But there are many other hazards for our programs...

NULL / UNDEFINED

But there are many other hazards for our programs...

NULL / UNDEFINED

EXCEPTIONS

But there are many other hazards for our programs...

NULL / UNDEFINED

EXCEPTIONS

INVARIANTS

As you probably guessed, these are also handled by the type system.

"Algebraic data types"

"Discriminated union types"

"Union types"

Similar concept to an `enum` that you might have seen in other languages.

EXAMPLE

```
type Bool   -- type
  = True    -- constructor or 'case'
  | False   -- constructor or 'case'
```

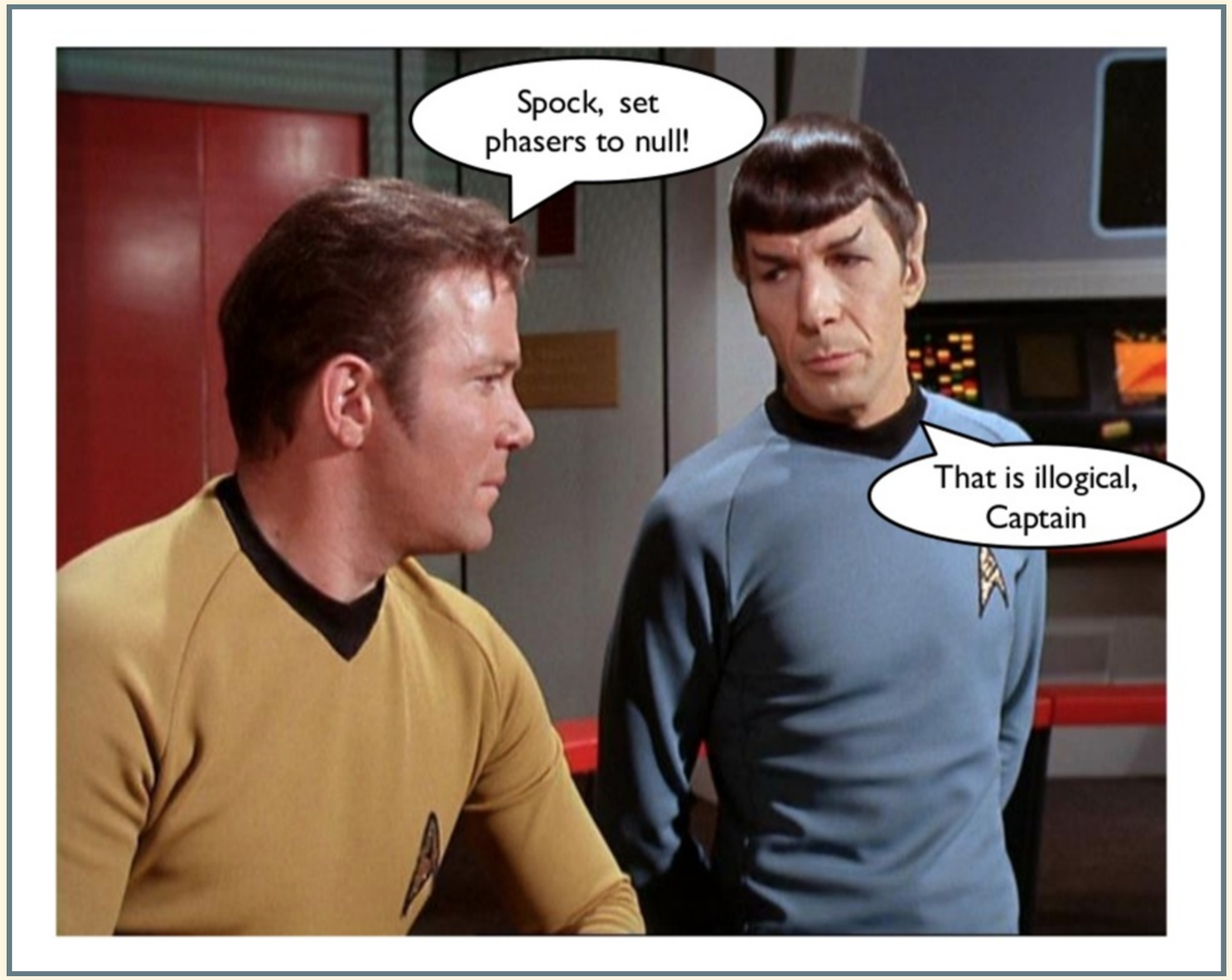
NOT a constructor in the OO sense. These are values, and the compiler will enforce that all possible cases of a union type are handled in your code.

Create your own for modeling your data...

```
type GamePhase
  = Ready
  | InProgress
  | Won
  | Lost
```

(we'll get back to this when we talk about 'invariants')

NULL / UNDEFINED



Martin Trojer
@martintrojer

undefined is not a function

It is known.

```
List.head ["Alice", "Bob", "Eve"]
> ...?

List.head []
> ...?
```

Let's try....

Maybe is a union type...

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

Represents values that may or may not exist.

The compiler forces you to handle each case...

```
names = ["Alice", "Bob", "Eve"]
first = List.head names

case first of
  Nothing
    -- ... the list was empty

  Just value ->
    -- ... succeed with value
```

Sometimes you just provide a default...

```
names = ["Alice", "Bob", "Eve"]
first = Maybe.withDefault "NONE" (List.head names)
```

Use Maybe in your data models:

```
type alias PersonName =
    { first : String        -- required
    , middle : Maybe String -- optional
    , last : String         -- required
    }
```

If a `Maybe` is not handled, your code **will not compile**.

http://package.elm-lang.org/packages/elm-lang/core/latest/Maybe

An **entire category** of

FUTURE BUGS

...is wiped from existence.

NULL / UNDEFINED

EXCEPTIONS

INVARIANTS

EXCEPTIONS

Search

[javascript] uncaught exception

search

1,761 results

relevance newest votes active

JavaScript (not to be confused with Java) is a dynamic, weakly-typed language used for client-side as well as server-side scripting. Use this tag for questions regarding ECMAScript and its various dialects/implementations (excluding ActionScript and Google-Apps-Script). Unless another tag for a ...

learn more… top users synonyms (12) javascript jobs

7 votes
1 answer

Q: Javascript error uncaught exception

I am using a **javascript** file called pull.js. It is using for pulldown refresh in Ipad but , when I use that other jquery and **javascript** stop working? It is giving the following **uncaught exception** … error : " Error: **uncaught exception**: [**Exception**... "Not enough arguments" nsresult: "0x80570001 (NS_ERROR_XPC_NOT_ENOUGH_ARGS)" location: "JS frame :: pull.js :: anonymous :: line 26 …

javascript jquery ios

asked Apr 24 '13 by Amar Banerjee

1 vote
3 answers

Q: Uncaught exception in firefox

I'm getting an **uncaught exception** in firefox. ([12:53:36.595] **uncaught exception**: Syntax error, unrecognized expression: input[id^=ctl00_ContentBody_cblAtmDebitCards) This is in the **javascript** of my … page that im writing in C# [12:53:36.595] **uncaught exception**: Syntax error, unrecognized expression: input[id^=ctl00_ContentBody_cblAtmDebitCards This is the whole call $.validator.addMethod …

c# javascript jquery

asked Nov 1 '13 by TheDizzle

4 votes

Q: HighCharts uncaught exception

="text/**javascript**" src="test.js"></script> tag after my tag, otherwise the **uncaught exception** will be showing even I put the <script> tag in <head>. I never put the script … I am trying to run

Exceptions are **lifted** into the type system just like **side effects** and **nulls**.

`Result error value`

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

A `Result` is either `Ok` meaning the computation succeeded, or it is an `Err` meaning that there was some failure.

The Elm compiler will ensure `Result` is used for any operation that might fail.

- conversions
- accessing http
- accessing storage
- etc...

```elm
numberString = "1664"
result = String.toInt numberString

case result of
  Err msg ->
    -- ... fail with message

  Ok value ->
    -- ... succeed with value
```

Again, you can provide a default...

```elm
numberString = "This ain't right"
result = Result.withDefault 0 (String.toInt numberString)

-- alternate syntax using the backwards pipe (fewer parens)
result = Result.withDefault 0 <| String.toInt numberString

-- alternate syntax using forwards pipe
result =
  numberString
    |> String.toInt
    |> Maybe.withDefault 0
```

Say goodbye to **runtime exceptions**...

NULL / UNDEFINED

EXCEPTIONS

INVARIANTS

So what is an invariant anyway?

Quesion: What kind of person would model a light bulb like this?

```
// JavaScript model of a light bulb...

var bulb = {
  on: true,
  off: false
}
```

Answer: Only a maniac.

```
var bulb = {
  on: true,
  off: true
}
```

It's only a matter of time before we get into an illegal state.

This is a silly example of an

INVARIANT

When you start looking you start to

NOTICE INVARIANTS

all over your code

Simple model for a Contact

```
type alias Contact =
    { name : String
    , email : String
    , address : String
    }
```

NEW REQUIREMENTS

- Email and address are now optional
- But a Contact must have **at least one contact method**

```
type alias Contact =
    { name : String
    , email : String
    , address : String
    }
```

We could make `Maybe` to make them both optional...

```
type alias Contact =
    { name : String
    , email : Maybe String
    , address : Maybe String
    }
```

But it's still possible to create a Contact in an invalid state:

```
dude =
  { name = "Jeffrey Lebowski"
  , email = Nothing
  , address = Nothing
  }
-- we're supposed to have at least one contact method...
```

Valid Option 1

```
type ContactMethod
    = EmailOnly String
    | AddressOnly String
    | EmailAndAddress (String, String) -- we use a Tuple
```

```
type alias Contact =
    { name : String
    , contactMethod : ContactMethod
    }
```

Try it out...

```
dude =
  { name = "Jeffrey Lebowski"
  , contactMethod = EmailOnly "el_duderino@yahoo.com"
  }
```

```
dude =
  { name = "Jeffrey Lebowski"
  , contactMethod = AddressOnly "11304 Malibu Heights"
  }
```

```
dude =
  { name = "Jeffrey Lebowski"
  , contactMethod =
      EmailAndAddress ("el_duderino@yahoo.com", "11304 Malibu Heights")
  }
```

Valid Option 2

```
type ContactMethod
  = Email String
  | Address String

type alias Contact =
  { name : String
  , primaryContact : ContactMethod
  , secondaryContact : Maybe ContactMethod
  }
```

Let's try it out...

```
dude =
  { name = "Jeffrey Lebowski"
  , primaryContact = Email "el_duderino@yahoo.com"
  , secondaryContact = Nothing
  }
```

```
dude =
  { name = "Jeffrey Lebowski"
  , primaryContact = Address "11304 Malibu Heights"
  , secondaryContact = Nothing
  }
```

```
dude =
  { name = "Jeffrey Lebowski"
  , primaryContact = Email "el_duderino@yahoo.com"
  , secondaryContact = Just "11304 Malibu Heights"
  }
```

Option 3

```
type ContactMethod
  = Email String
  | Address String
```

```
type alias Contact =
  { name : String
  , contactMethod : ContactMethod
  , alternateContactMethods : List ContactMethod
  }
```

With our first model, we had to rely on

PROGRAMMER CORRECTNESS

```
type alias Contact =
    { name : String
    , email : Maybe String
    , address : Maybe String
    }
-- it's possible for email and address to both be Nothing
```

WE COULD LIVE WITH THIS.

WE COULD LIVE WITH THIS.

MOST OF US DO LIVE WITH THIS.

WE COULD LIVE WITH THIS.

MOST OF US DO LIVE WITH THIS.

WE JUST COMMUNICATE THE RULES TO EVERYONE.

WE COULD LIVE WITH THIS.

MOST OF US DO LIVE WITH THIS.

WE JUST COMMUNICATE THE RULES TO EVERYONE.

REMEMBER TO TELL THE NEW GUY.

WE COULD LIVE WITH THIS.

MOST OF US DO LIVE WITH THIS.

WE JUST COMMUNICATE THE RULES TO EVERYONE.

REMEMBER TO TELL THE NEW GUY.

MAYBE WRITE SOME COMMENTS IN OUR CODE.

WE COULD LIVE WITH THIS.

MOST OF US DO LIVE WITH THIS.

WE JUST COMMUNICATE THE RULES TO EVERYONE.

REMEMBER TO TELL THE NEW GUY.

MAYBE WRITE SOME COMMENTS IN OUR CODE.

WE MIGHT EVEN WRITE TESTS!

~~WE COULD LIVE WITH THIS.~~

~~MOST OF US DO LIVE WITH THIS.~~

~~WE JUST COMMUNICATE THE RULES TO EVERYONE.~~

~~REMEMBER TO TELL THE NEW GUY.~~

~~MAYBE WRITE SOME COMMENTS IN OUR CODE.~~

~~WE MIGHT EVEN WRITE TESTS!~~

Leverage the compiler to

ENFORCE OUR BUSINESS RULES

Tests are good

IMPOSSIBLE IS BETTER

NULL / UNDEFINED

EXCEPTIONS

INVARIANTS

Host of tools for improving your code

null / undefined	`Maybe a`
exceptions	`Result error value`
invariants	types + compiler
side effects	`Cmd msg`
mutating the dom manually	virtual dom
mutable shared state	all data immutable
state was spread out	centralized state
multidirectional data flow	unidirectional data flow
silent type errors	strong type system

PART 1 - BACKGROUND

PART 2 - INTRO TO ELM

PART 3 - WHY ELM IS DIFFERENT

PART 4 - TYPE SYSTEM

PART 5 - UP AND RUNNING

Installing and getting going

- `npm i -g elm`
- `mkdir my-proj && cd my-proj`
- `elm package install`
- `atom Main.elm`

ELM PACKAGE MANAGER

No user will ever get a breaking API change in a patch version.

ENFORCED SEMANTIC VERSIONING

```
elm package diff elm-lang/core 3.0.0 4.0.0
```

COME BACK MONDAY

compiler helps you pick up right where you left off

INTERRUPTIONS?

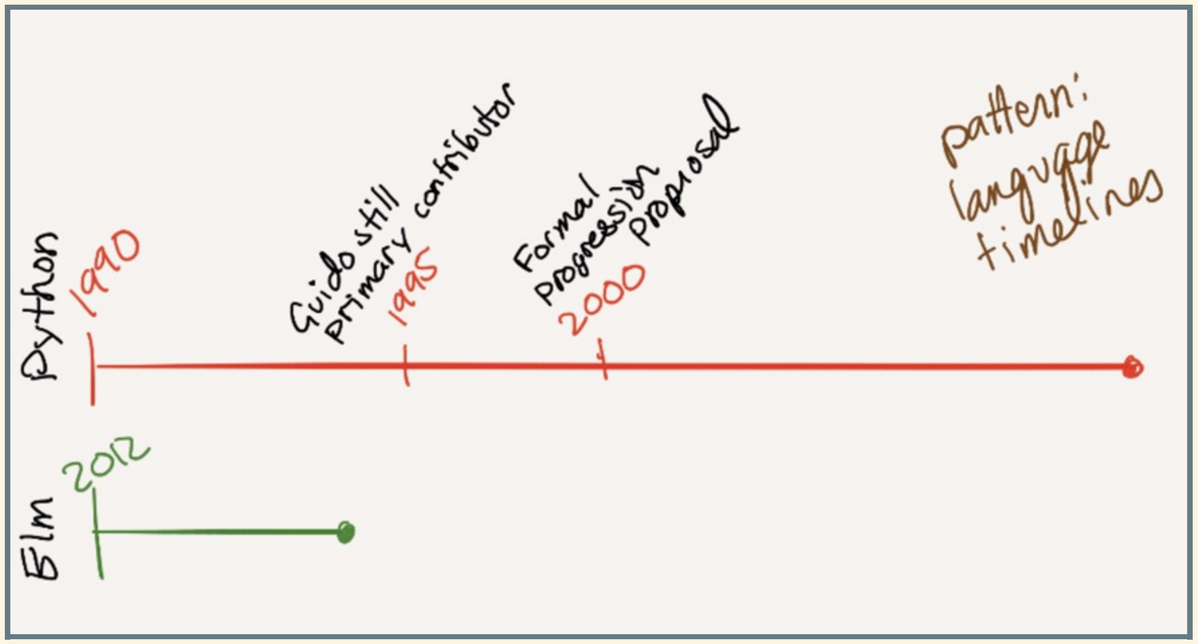
plates stay spinning

"COMPILERS AS THERAPISTS, OR WHY ELM IS GOOD FOR ADHD"

Luke Westby

https://www.youtube.com/watch?v=wpYFTG-uViE

Timeline (compared to python)



"JUST DO A GOOD JOB"

- Advice to Evan Czaplicki, *Elm's creator*, from Guido van Rossum, *Python's creator*.

AT COMPILE TIME

- It is impossible to mutate data
- It is impossible to not handle "null/undefined"
- It is impossible to not to handle exceptions
- It is impossible to have runtime type errors
- It is impossible to write an impure function
- It is impossible to mix side effects with other logic
- It is impossible to not handle all cases of a union type

Things I didn't have time for (but we can talk about):

- JS interop (the safe way)
- Integrate with existing projects
  - Backbone --> React
  - React --> Elm
  - integration with redux
- Use for just a portion of your web application (grow from there)

PEOPLE

- Evan Czaplicki (Elm creator) @czaplic
- Kris Jenkins (London) @krisajenkins
- Richard Feldman (NoRedInk) @rtfeldman
- Simon Peyton Jones (Microsoft Research) Google him
- Yaron Minsky (Jane Street) @yminksy

TOOLS

- elm-format - code formatter
- elm-oracle - auto-completion
- Reveal JS - Node-based presentation tool
- Reveal-MD - Reveal JS + Markdown on the command line
- https://atom.io/packages/elm-format
- https://atom.io/packages/language-elm
- https://atom.io/packages/linter-elm-make

LINKS

- Join FCIP on Slack!
- Original thesis on Elm - Evan Czaplicki
- What is Functional Programming - Kris Jenkins
- Haskell is Useless - Simon Peyton Jones
- A Tool for Thought - David Nolan
- Designing with types: Making illegal states unrepresentable - Scott Wlaschin *(F#)*
- The Wrong Abstraction - Sandi Metz
- Effective ML (Vimeo) - Yaron Minksy
- Compilers as Therapists - Luke Westby

FIN

comments or questions?