Elm: Building Reactive Web Apps

Cheat Sheets

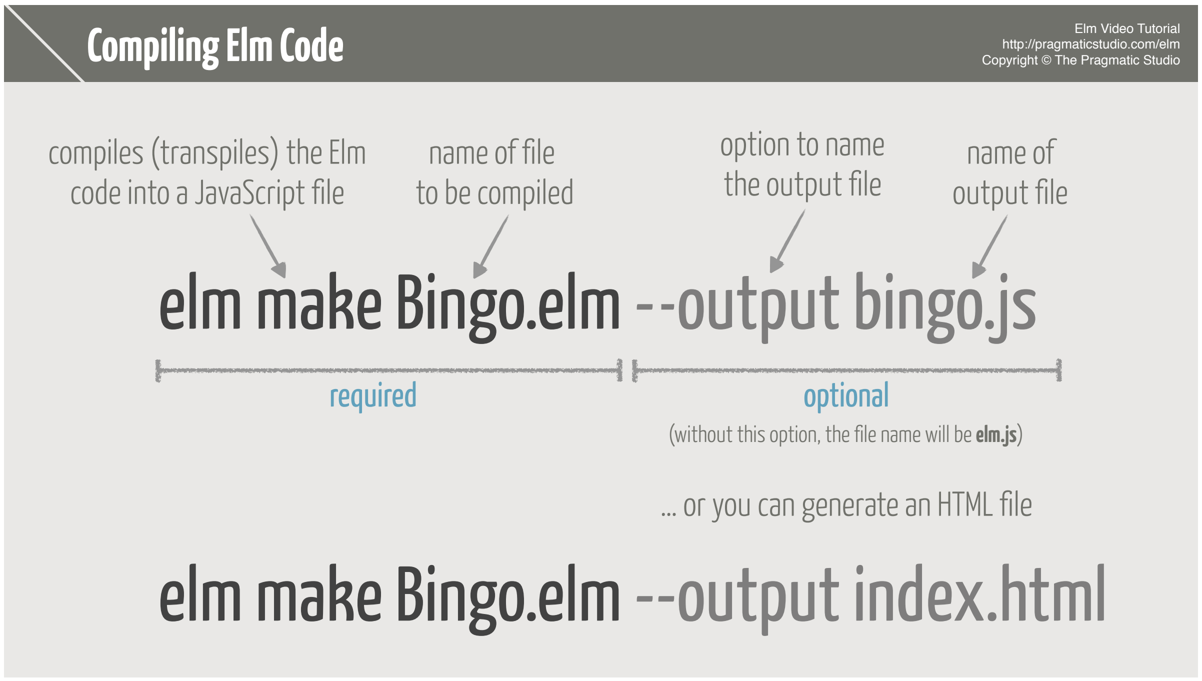
The Pragmatic Studio

Craft Reactive Web Apps, The Elm Way

Elm is a functional programming language that compiles to JavaScript and runs in the browser, designed and developed by Evan Czaplicki. Elm syntax is clean and readable. Refactoring and testing is a breeze. The best-of-class Elm compiler always generates reliable code, so runtime exceptions are a thing of the past. And Elm can render HTML with blazing speed. But what we love most about Elm is that you can actually build practical stuff with it quickly!

Learn Elm With Us, The Pragmatic Way

- 20 downloadable videos so you learn how to build feature-rich Elm web apps step-by-step
- Example code and setup instructions so you can prepare your development environment, follow along with the videos, and experiment on your own
- Course designed specifically for experienced web developers who want to have some fun learning Elm in the context of building a web app



Compiling Elm Code

compiles (transpiles) the Elm code into a JavaScript file

name of file to be compiled

option to name the output file

name of output file

```
elm make Bingo.elm --output bingo.js
```

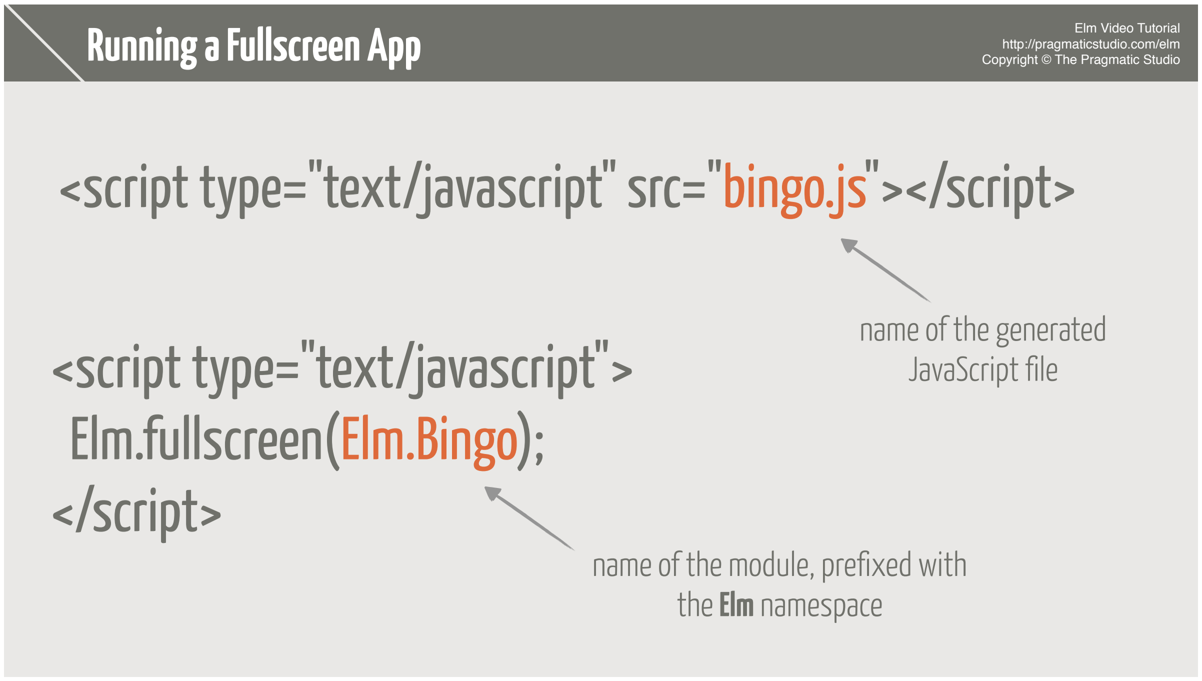
required

optional

(without this option, the file name will be **elm.js**)

... or you can generate an HTML file

```
elm make Bingo.elm --output index.html
```



Running a Fullscreen App

```
<script type="text/javascript" src="bingo.js"></script>
```

name of the generated JavaScript file

```
<script type="text/javascript">
 Elm.fullscreen(Elm.Bingo);
</script>
```

name of the module, prefixed with the **Elm** namespace

Installing Packages

elm package install

If an **elm-package.json** file exists, this installs all the package dependencies.
Otherwise, this installs the **elm-core** package and creates an **elm-package.json** file.

elm package install evancz/elm-html

Installs the specified package and updates the **elm-package.json** file

Using the REPL

elm repl

import required modules

```
> import String
```

easy way to get the function signature

```
> String.toUpper
<function: toUpper> : String -> String
```

```
> String.toUpper "Bingo!"
"BINGO!" : String
```

enter Elm expressions and get the resulting value and type

```
> String.toUpper \
| "Bingo!"
"BINGO!" : String
```

use a backslash (\) for multi-line expressions

```
> :exit
```

* requires **node.js** to evaluate the generated JavaScript

Calling a Function

$$\underbrace{\text{String.repeat}}_{\text{function name}}\quad \underbrace{3}_{\text{1st arg}}\quad \underbrace{\text{“Bingo!”}}_{\text{2nd arg}}$$

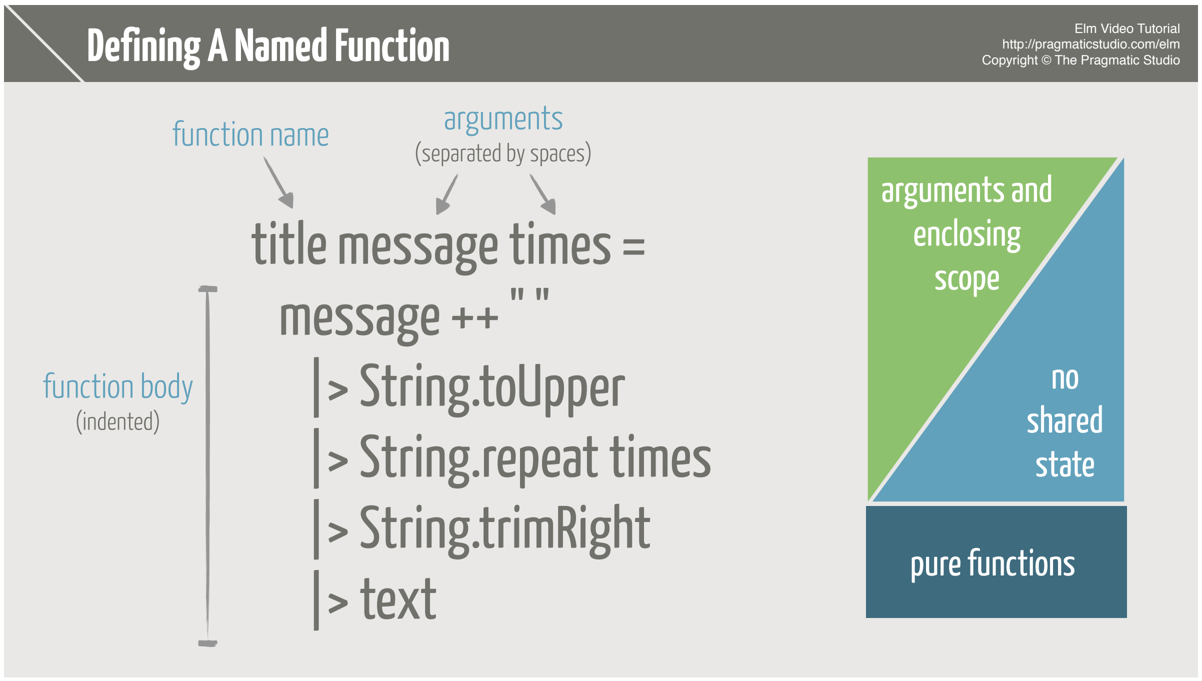
The function arguments are separated from the function name by a single space. And function arguments are separated by a single space, too. **No commas!**

Pipe Operator

```
main =
 text (String.repeat 3 (String.toUpper "Bingo!"))
```

```
main =
 "Bingo!"
    |> String.toUpper
    |> String.repeat 3
    |> text
```

These do the **same** thing!



Defining A Named Function

Elm Video Tutorial
http://pragmaticstudio.com/elm
Copyright © The Pragmatic Studio

function name

arguments
(separated by spaces)

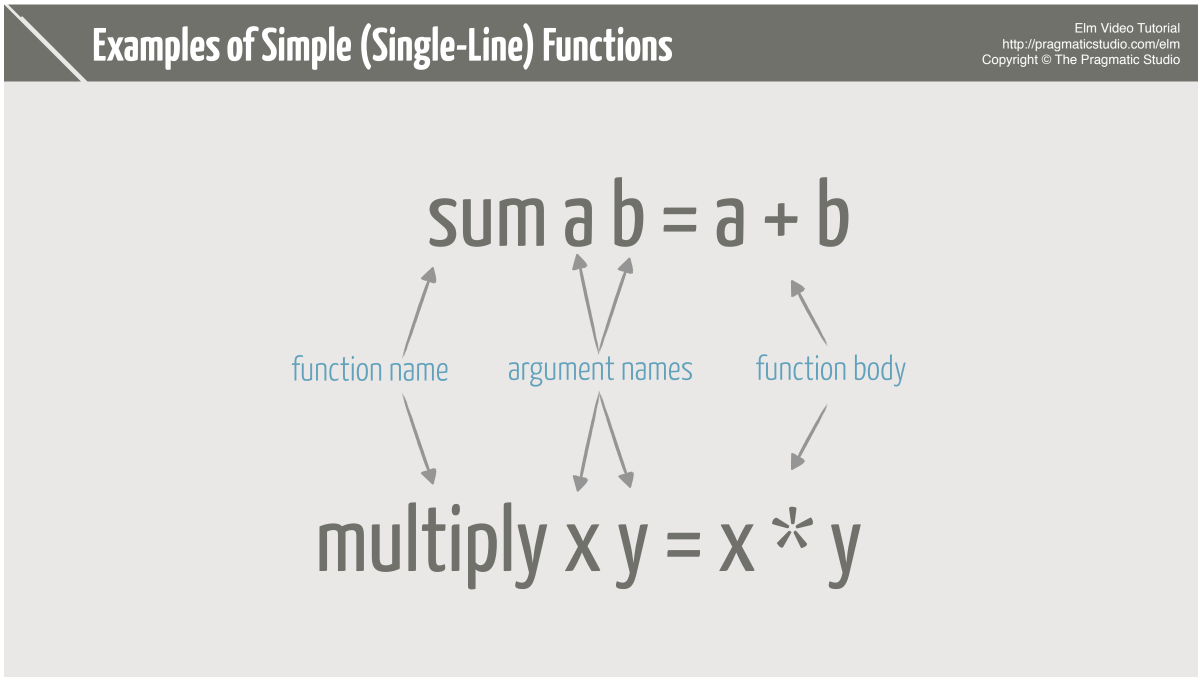
function body
(indented)

```
title message times =
  message ++ " "
    |> String.toUpper
    |> String.repeat times
    |> String.trimRight
    |> text
```

arguments and enclosing scope

no shared state

pure functions



Examples of Simple (Single-Line) Functions

Elm Video Tutorial
http://pragmaticstudio.com/elm
Copyright © The Pragmatic Studio

```
sum a b = a + b
```

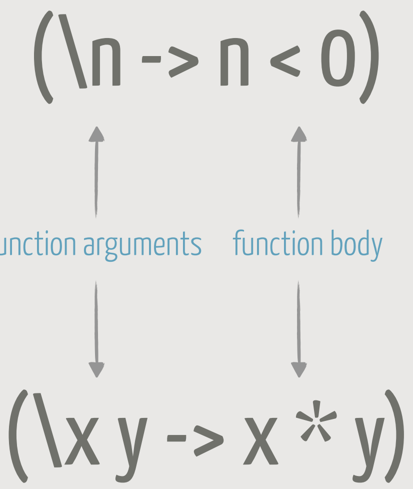
function name

argument names

function body

```
multiply x y = x * y
```

Anonymous Functions



```
(\n -> n < 0)
```

```
(\x y -> x * y)
```

Can be passed any time a function argument is expected. For example:

```
List.filter (\n -> n < 0) [-2..3]

List.map (\n -> n * 2) [1..3]
```

Importing Modules

import String

no functions are introduced from the module into the current namespace	**all** functions **must be qualified** with the module name	**String.toUpper**

import String exposing (..)

introduces **all** the functions in the module into the current namespace	can use **all** the functions **without qualification**	**toUpper**

import String exposing (toUpper, repeat, trimRight)

introduces **only the specified** functions in the module into the current namespace	can use **all** the functions **without qualification**	**toUpper**

Lists

A list holds a collection of related values separated by commas and enclosed in square brackets. All the values in a list must have the **same type**.

Three ways to create the **same** list:

```
[1, 2, 3, 4]

[1..4]

1 :: [2, 3, 4]
```

Generating Basic HTML

```
tag [ attributes ] [ children ]
```

```
h1 [ id "logo", class "classy" ] [ text "Welcome!" ]

p [ ] [ text "Elm generates HTML and CSS..." ]

a [ href "https://pragmaticstudio.com" ] [ text "The Pragmatic Studio" ]
```

Generating Nested HTML

```
ul [ id "todos" ]
 [ li [ ] [ text "A" ],
   li [ ] [ text "B" ],
   li [ ] [ text "C" ]
 ]
```

```
div [ id "container" ]
 [ pageHeader,
   pageFooter
 ]
```

Preferred style is to list child elements on separate lines

Records

A record is a collection of key/value pairs, similar to objects in JavaScript or hashes in Ruby. Records, however, are **immutable**.

field name ↓ field value ↓

```
{ phrase = “Reactive”,
  points = 150,
  wasSpoken = False
}
```

Accessing Record Field Values

```
entry =
  { phrase = “Reactive”,
    points = 150,
    wasSpoken = False
  }
```

entry.phrase →	"Reactive"
entry.points →	150
entry.wasSpoken →	False

Record Access Syntactic Sugar

Accessing a field value this way is equivalent to using an anonymous function:

.phrase entry	(\e -> e.phrase)
.points entry	(\e -> e.points)
.wasSpoken entry	(\e -> e.wasSpoken)

Those anonymous functions can then be used as function arguments. For example:

```
List.map .phrase entries     List.sortBy .points entries     List.filter .wasSpoken entries
```

Updating Records

clones the **entry** record and updates its **points** field

```
{ entry | points <- 500 }
```

```
{ entry | points <- 500, wasSpoken <- True }
```

You can update multiple fields, separating each update by a comma

* ALWAYS returns a new record!

Union Types (aka Enumerations)

Specifies the kinds of actions that can be performed on a model

```
type Action = Sort | Mark Int | Delete Int
```

name

possible values separated by a vertical bar

* it's just data — doesn't actually DO anything

Pattern Matching Actions

```
update action model =
 case action of
  NoOp ->
   model

  Sort ->
   { model | entries <- List.sortBy .points model.entries }
```

case tries to match the value of action against patterns defined after the of keyword

if the pattern is matched, the value of the expression after -> becomes the result of the case expression

Read it like: “If you see this pattern, evaluate this -> expression!”

Let Expressions

```
let
  <definitions>
in
  <expression>
```

```
Delete id ->
 let
  remainingEntries = List.filter (\e -> e.id /= id) model.entries
 in
  { model | entries <- remainingEntries }
```

this function is “local” to the enclosing scope

value of the expression defined after the in keyword is the result of the let expression

can use definitions defined between let and in

If Expressions

```
if <condition> then <result1> else <result2>
```

```
if True then "yup" else “nope”    →  "yup"
if False then "yup" else “nope”   →  "nope"
if 1 then "yup" else “nope”       →  Error!
```

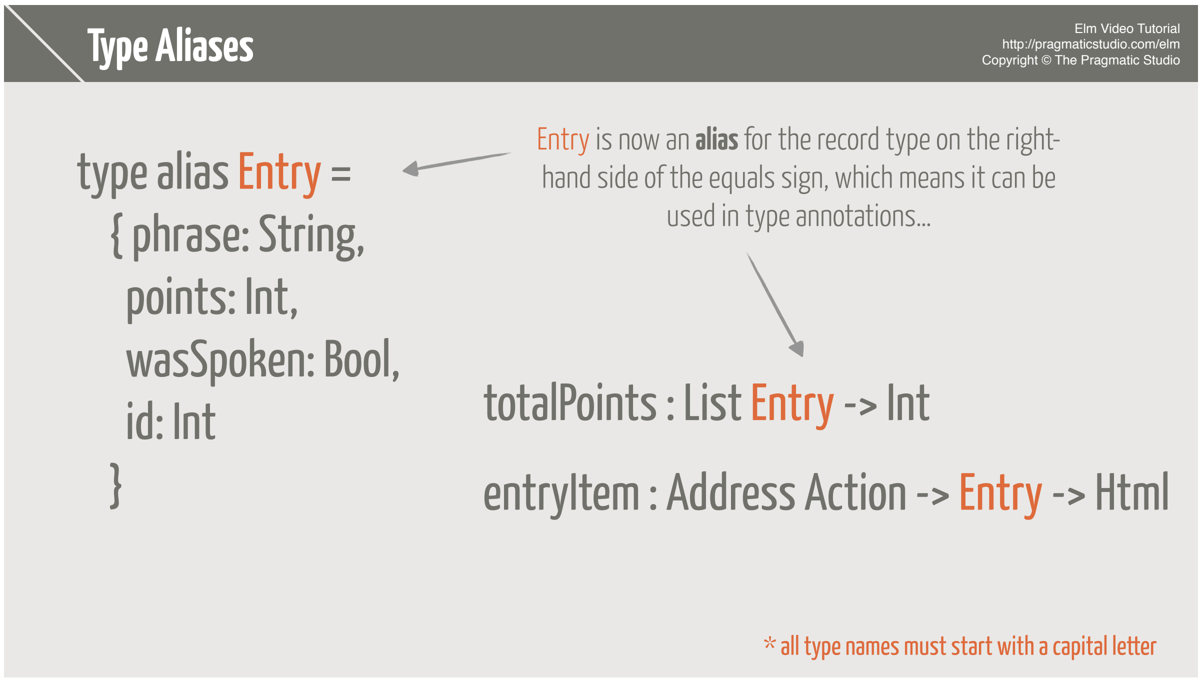
* Elm doesn’t have the notion of “truthiness”. The condition must evaluate to True or False, and nothing else. Thankfully, the compiler will catch this at compile time, so the error can't sneak into production!

Type Annotations

$$\underbrace{\text{update}}_{\text{function name}} : \underbrace{\text{Action}}_{\text{1st arg type}} \to \underbrace{\text{Model}}_{\text{2nd arg type}} \to \underbrace{\text{Model}}_{\text{return type}}$$

$$\text{map} : \underbrace{(a \to b)}_{\text{function that takes an } \mathbf{a} \text{ value and returns a } \mathbf{b} \text{ value}} \to \underbrace{\text{List } a}_{\text{list of } \mathbf{a} \text{ values}} \to \underbrace{\text{List } b}_{\text{returns a list of } \mathbf{b} \text{ values}}$$

* **a** and **b** are placeholders for any type



Type Aliases

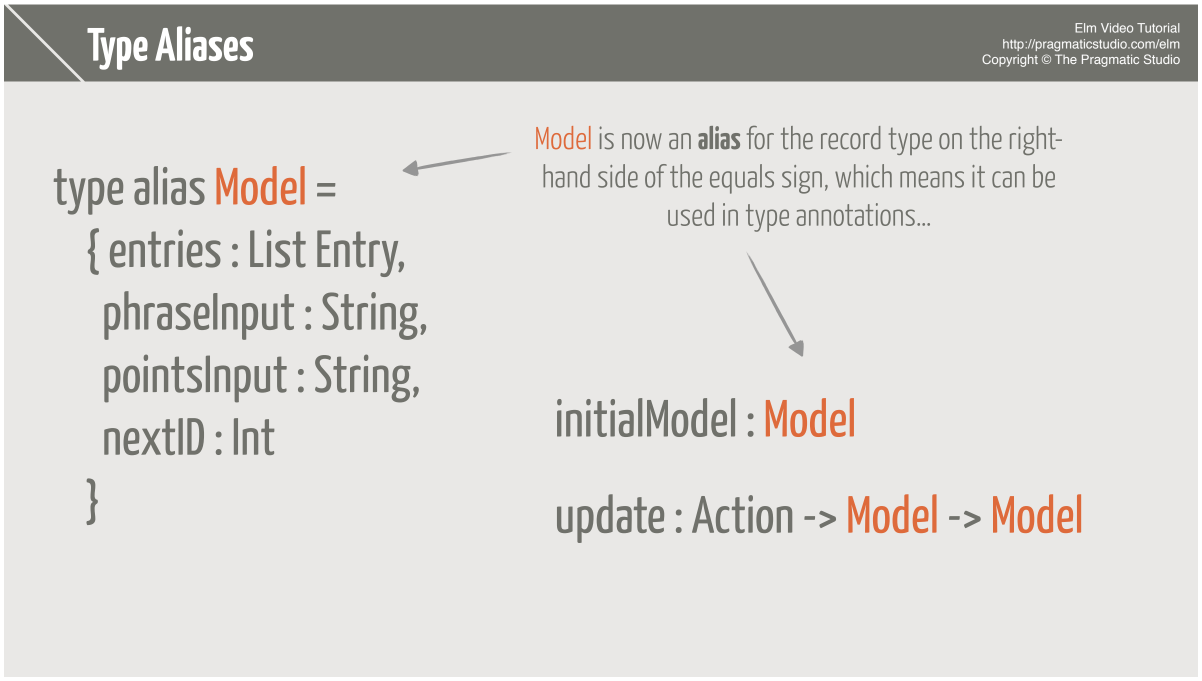
```
type alias Entry =
  { phrase: String,
    points: Int,
    wasSpoken: Bool,
    id: Int
  }
```

Entry is now an **alias** for the record type on the right-hand side of the equals sign, which means it can be used in type annotations...

```
totalPoints : List Entry -> Int

entryItem : Address Action -> Entry -> Html
```

* all type names must start with a capital letter



Type Aliases

```
type alias Model =
  { entries : List Entry,
    phraseInput : String,
    pointsInput : String,
    nextID : Int
  }
```

Model is now an **alias** for the record type on the right-hand side of the equals sign, which means it can be used in type annotations...

```
initialModel : Model

update : Action -> Model -> Model
```

Event Handlers

```
button [ class "sort", onClick address Sort ] [ text "Sort" ]
```

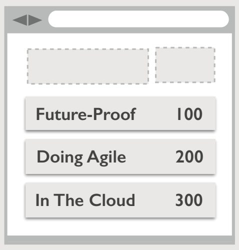
every event handler reports to an address

sends an action which is just data

```
button [ class "delete", onClick address (Delete entry.id) ] []
```

onClick simply returns a message, which is like an envelope that already has an address and the letter inside (the action), but hasn't yet been sent!

The Elm Architecture



```
main: Signal Html
main =
  StartApp.start { model = initialModel,
                   view = view,
                   update = update
                 }
```

MODEL

```
type alias Model =
  { entries : List Entry,
    phraseInput : String,
    pointsInput : String,
    nextID : Int
  }
```

UPDATE

```
type Action = Sort | Mark Int | Delete Int ...

update : Action -> Model -> Model
update action model =
  ...
```

VIEW

```
view : Address Action -> Model -> Html
view address model =
  ...
```