Follow Along

Clone this:

https://github.com/paulcoyle/yycjs-elm/

Paste code here:

http://elm-lang.org/try

Or, use elm-reactor if you've installed Elm

An Introduction to Functional Programming using Elm

Matt Hughes and Paul Coyle

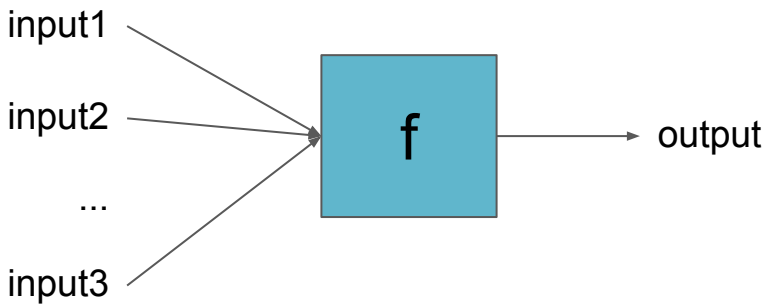
April 19, 2016

Image Credit: Jeff Smits
https://github.com/elm-lang/elm-lang.org/blob/master/resources/logo.svg

What is functional programming?

- Stateless functions, with explicit inputs and outputs

output = f(input1, input2, …, inputN)



What is a functional programming language?

- A language that makes it easy to write code without side-effects.
- Or… hard to write code *with* side-effects.
- Actively *hostile* to side-effects!

Why functional programming is awesome

Stateless functions

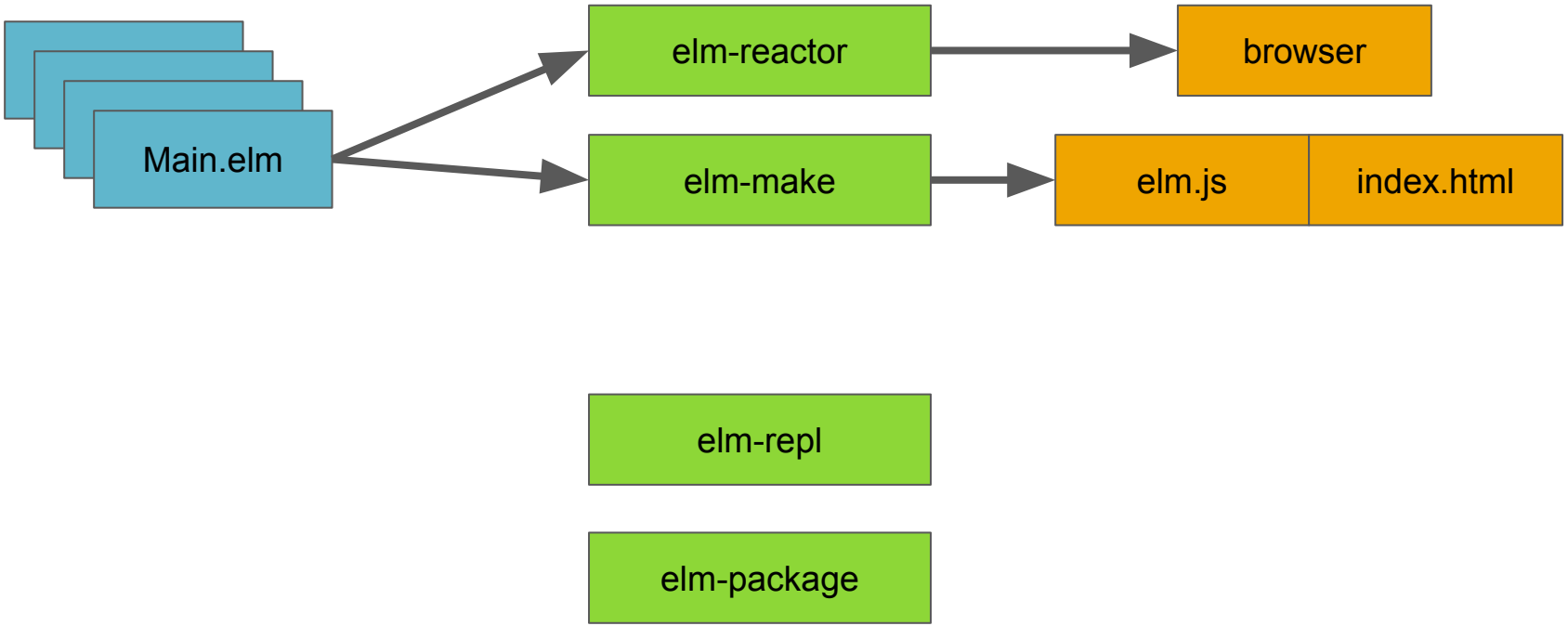
- Simple mental model of computation
- Reliable
- Easy to refactor

Image Credit: Randall Munroe
http://www.xkcd.com/1270/

What is Elm?

- A Functional Programming Language
- Compiles to JavaScript
- Provides functional methods of handling user input and other events
- Integrates with existing JavaScript apps

Elm’s tools


Main.elm
elm-reactor
browser
elm-make
elm.js
index.html
elm-repl
elm-package

Why Elm?

- Super easy to get started!
- Visual and interactive

http://elm-lang.org/try

- Try the examples:

http://elm-lang.org/examples

- Grab our code, paste it in, and play with it:

https://github.com/paulcoyle/yycjs-elm

McMaster University Computing and Software Outreach

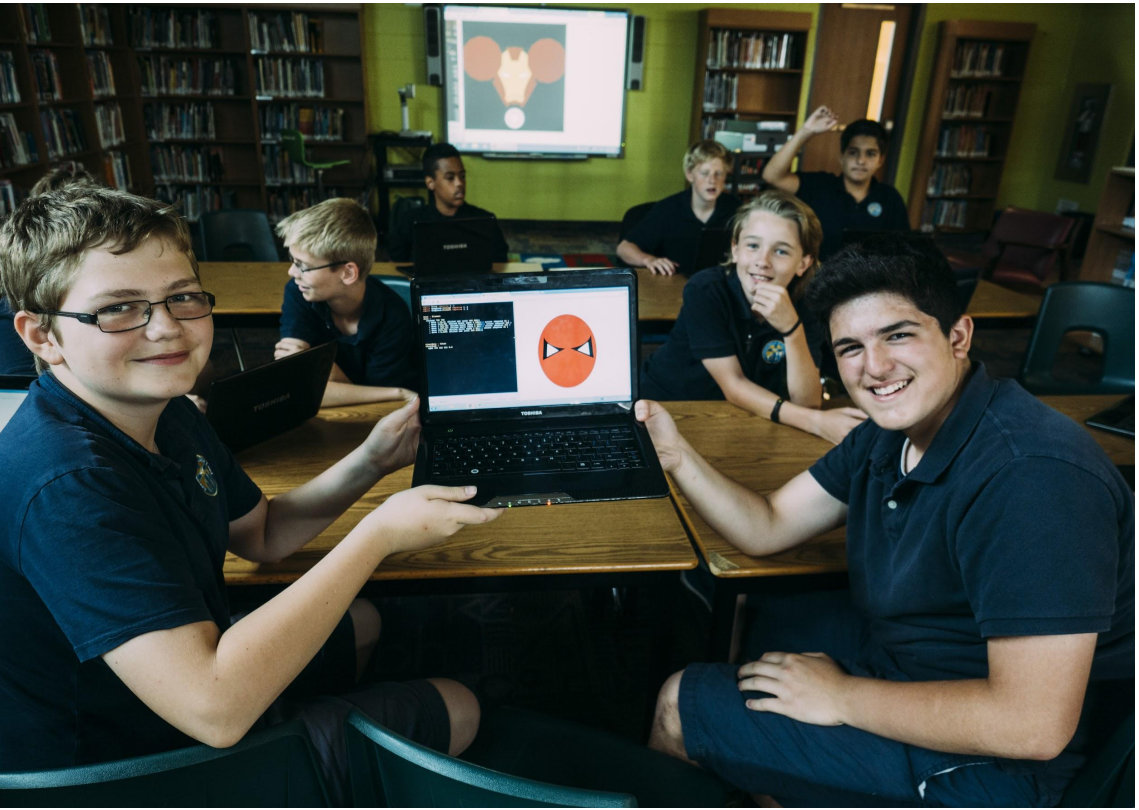


Image Credit: McMaster University
http://outreach.mcmaster.ca/updates/twelve.jpg
http://outreach.mcmaster.ca/updates/dpool.gif

DEADPOOL

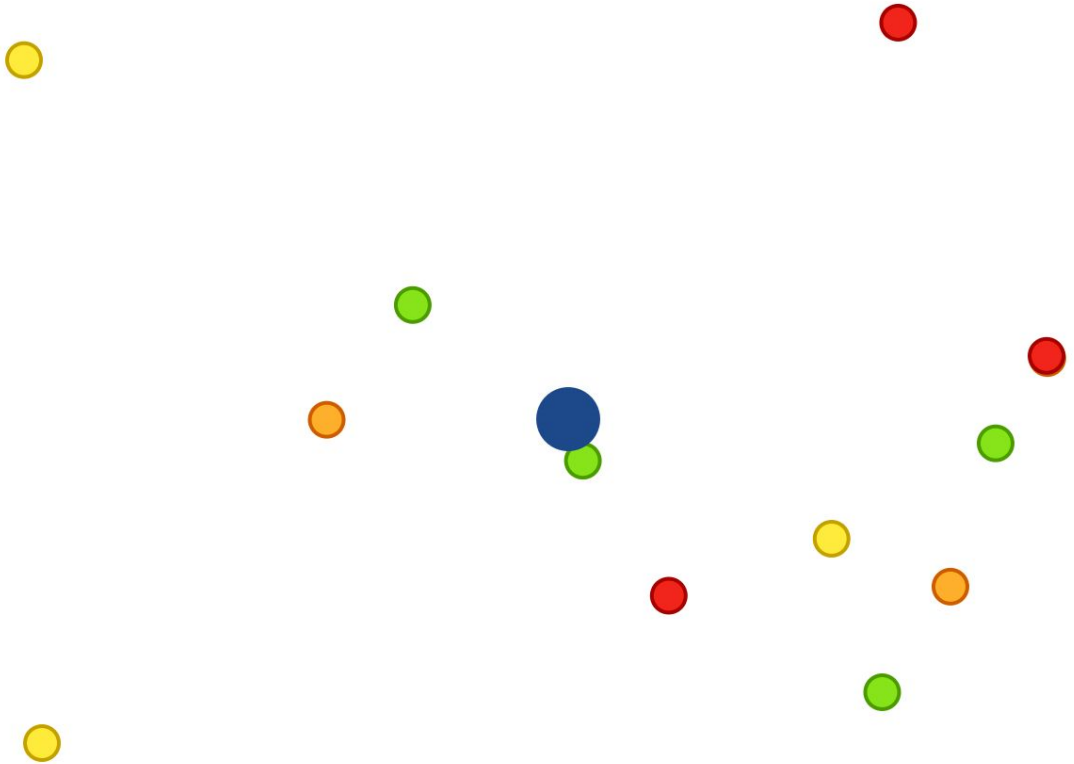
Created by Joey in Grade 8!

Image Credit: Josh Wills
https://twitter.com/josh_wills/status/422136541851312128
https://memegenerator.net/instance/44767782

A few more reasons to try Elm

- **World's friendliest error messages**
- Time travelling debugging
- Enforced semantically versioned libraries
- Enough features to get going, not enough to get lost.
- Tooling
- Strong type system
- Fast virtual DOM diffing and rendering library
- JavaScript interop
- No runtime exceptions

Let’s Build Something!



Step 1: Let’s get started!

http://elm-lang.org/examples/hello-html

```
import Html exposing (text)



main =
  text "Hello, World!"
```

The `text` function

text : String -> Html

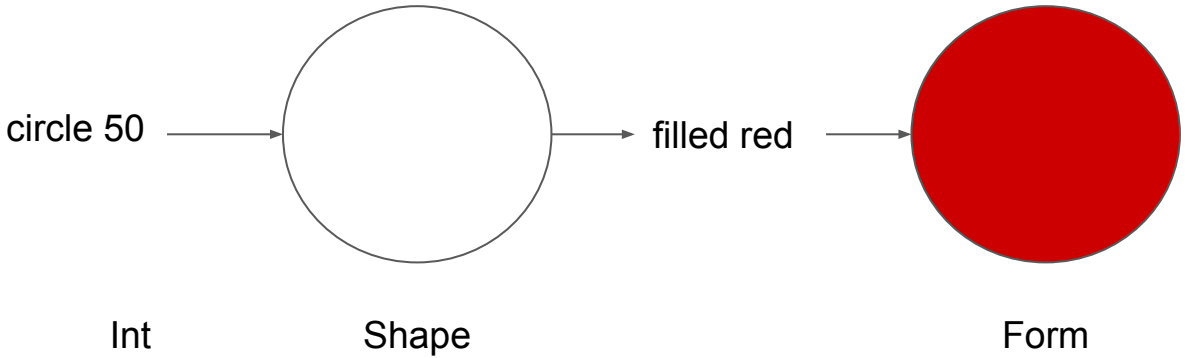
Just put plain text in the DOM. It will escape the string so that it appears exactly as you specify.

http://package.elm-lang.org/packages/evancz/elm-html/4.0.2/Html#text

Step 2: Let’s draw a picture

```
import Html exposing (text, fromElement)
import Graphics.Collage exposing (..)
import Color exposing (..)

main =
  fromElement (collage 320 240 [filled red (circle 50)])
```

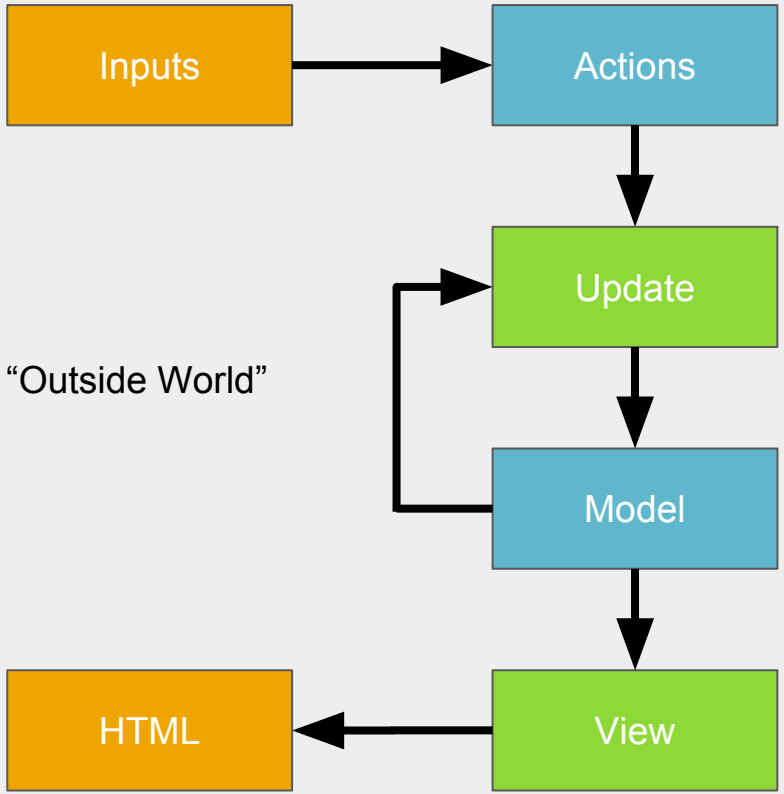
Thinking in types




```
circle : Int -> Shape

filled : Color -> Shape -> Form
```

“The Elm Architecture”

Elm’s Component Pattern (Simplified)

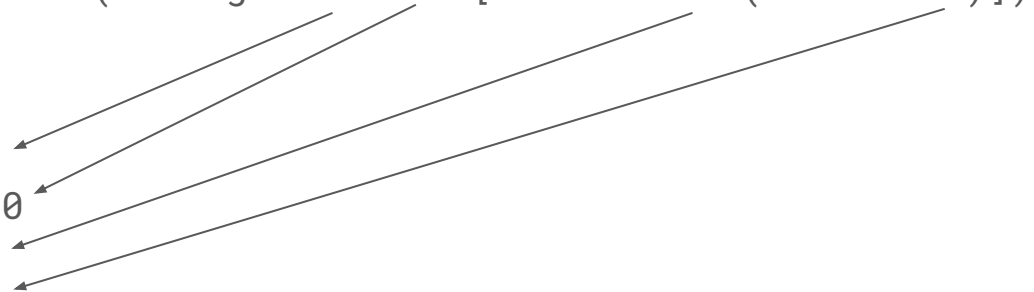


Step 3: StartApp.Simple

```
import Html exposing (text, fromElement)
import Graphics.Collage exposing (..)
import Color exposing (..)
import StartApp.Simple as StartApp

main =
  StartApp.start
    { model = {}
    , view = view
    , update = update
    }

update action model =
  model

view address model =
  Html.fromElement (collage 320 240 [filled red (circle 50)])
```

The Model

```
view address model =
  Html.fromElement (collage 320 240 [filled red (circle 50)])


model =
  { width = 320
  , height = 240
  , color = red
  , radius = 50
  }
```



Step 4: The view function

```
view address model =
  Html.fromElement
    (collage
      model.width
      model.height
      [filled model.color (circle model.radius)])


model =
  { width = 320
  , height = 240
  , color = red
  , radius = 50
  }
```

Step 5: Sending a message

```
type Action
  = MouseMove Int Int


view address model =
  Html.div
    [ onMouseMove address MouseMove ]
        [ Html.fromElement
            (collage
               model.width
               model.height
               [ filled model.color (circle model.radius) ]
            ) ]
```

onMouseMove

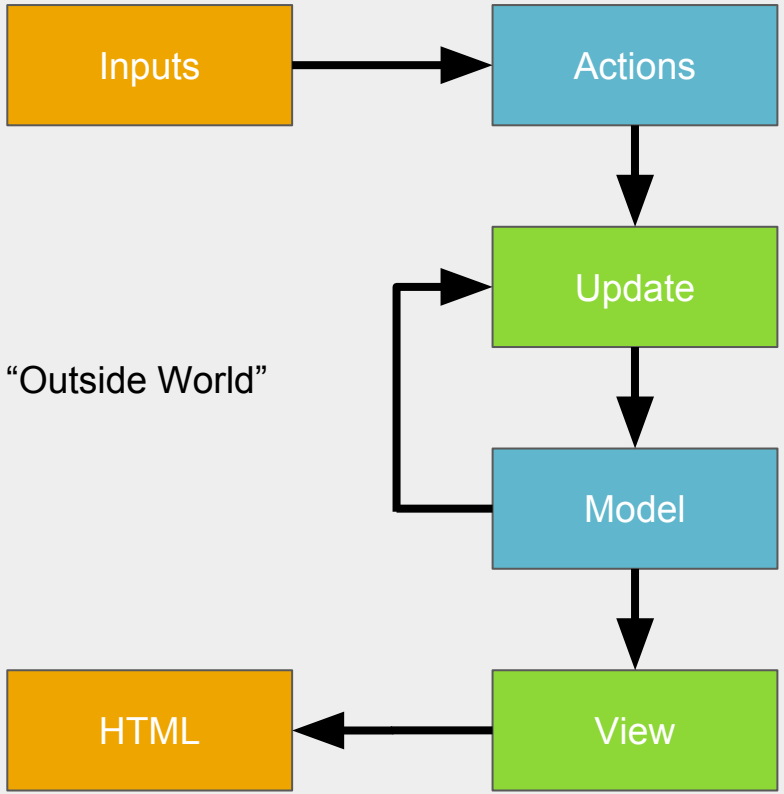
```
import Html.Events exposing (on)
import Json.Decode exposing ((:=), int, object2)

-- ... removed ...

onMouseMove
  : Signal.Address Action
  -> (Int -> Int -> Action)
  -> Html.Attribute
onMouseMove address f =
  on "mousemove"
    (object2 (\x y -> { x = x, y = y }) ("clientX" := int) ("clientY" := int))
    (\data -> message address (f data.x data.y))
```

Step 6: Updating the Model

```
model =
  { width = 320
  , height = 240
  , color = red
  , radius = 50
  , px = 0
  , py = 0
  }
```

“The Elm Architecture”

Elm’s Component Pattern (Simplified)



Step 6: The update function

```
update action model =
  case action of
    MouseMove x y ->
      { model
        | px = toFloat (x - model.width // 2)
        , py = toFloat (model.height // 2 - y) }
```

Step 6: The view function

```
view address model =
  Html.div
    [ onMouseMove address MouseMove ]
    [ Html.fromElement
        (collage
          model.width
          model.height
          [ move
              (model.px, model.py)
              (filled model.color (circle model.radius)) ]
        )
    ]
```

Step 7: Scoring points

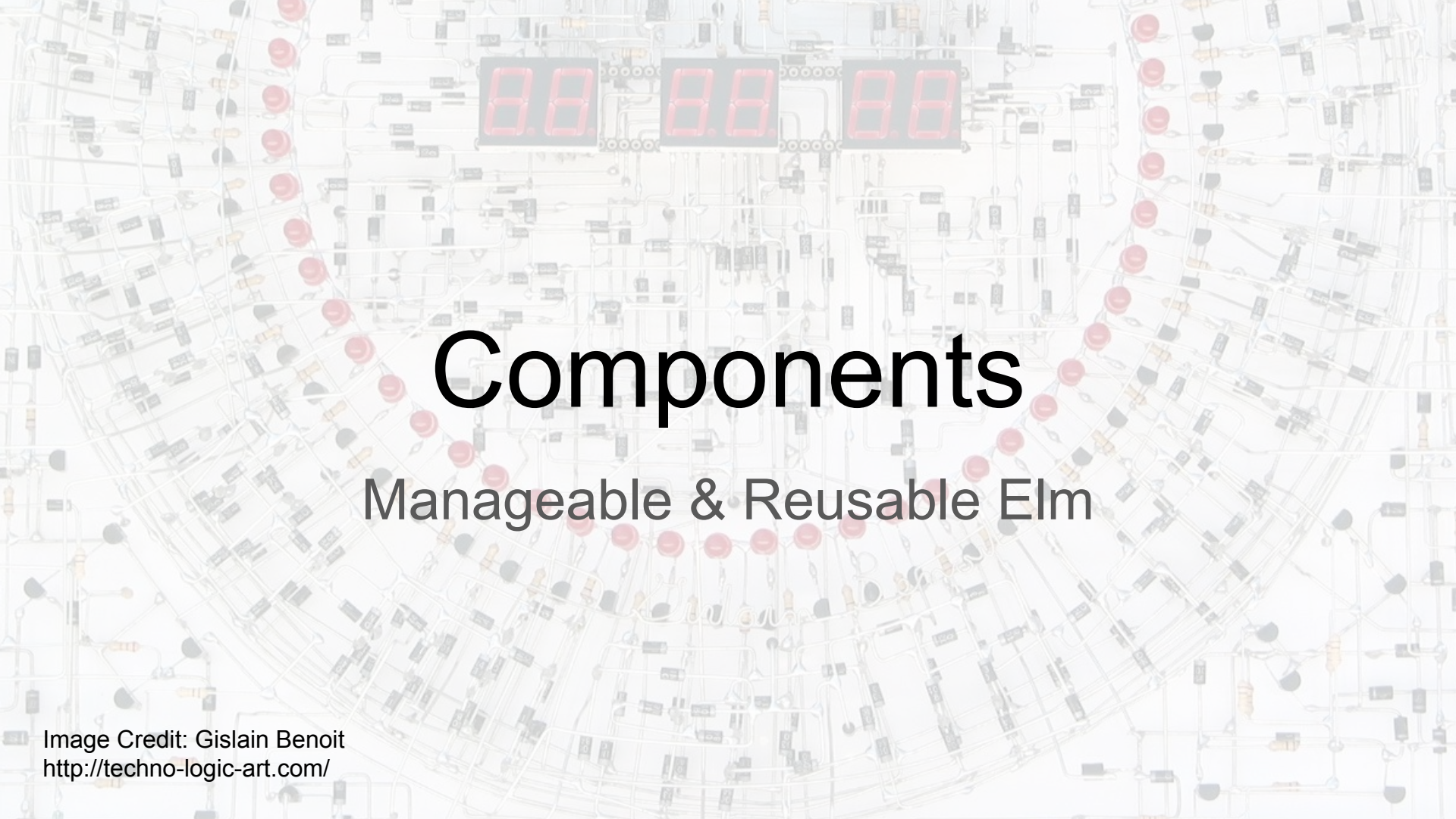
```
model =
  { width = 320
  , height = 240
  , color = red
  , radius = 50
  , px = 0
  , py = 0
  , food = [ {px = -140, py = -100, radius = 10}
           , {px = 140, py = -100, radius = 10}
           , {px = -140, py = 100, radius = 10}
           , {px = 140, py = 100, radius = 10}
           ]
  , score = 0
  }
```

Step 7: view function

```
view address model =
  Html.div
    [ onMouseMove address MouseMove ]
    [ Html.fromElement
        (collage model.width model.height
          (move (model.px, model.py)
            (filled model.color (circle model.radius))
          :: List.map viewFood model.food)
        )
    , Html.div [] [ Html.text ("Score: " ++ (toString model.score)) ]
    ]

viewFood food =
  circle food.radius
    |> filled blue
    |> move (food.px, food.py)
```

Step 7: update function

```
update action m =
  case action of
    MouseMove x y ->
      let
        notEaten f = not (collided m.px m.py m.radius f.px f.py f.radius)

        remainingFood =
          List.filter notEaten m.food

        points = ((List.length m.food) - (List.length remainingFood)) * 100
      in
      { m
        | px = toFloat (x - m.width // 2)
        , py = toFloat (m.height // 2 - y)
        , food = remainingFood
        , score = m.score + points
      }
```



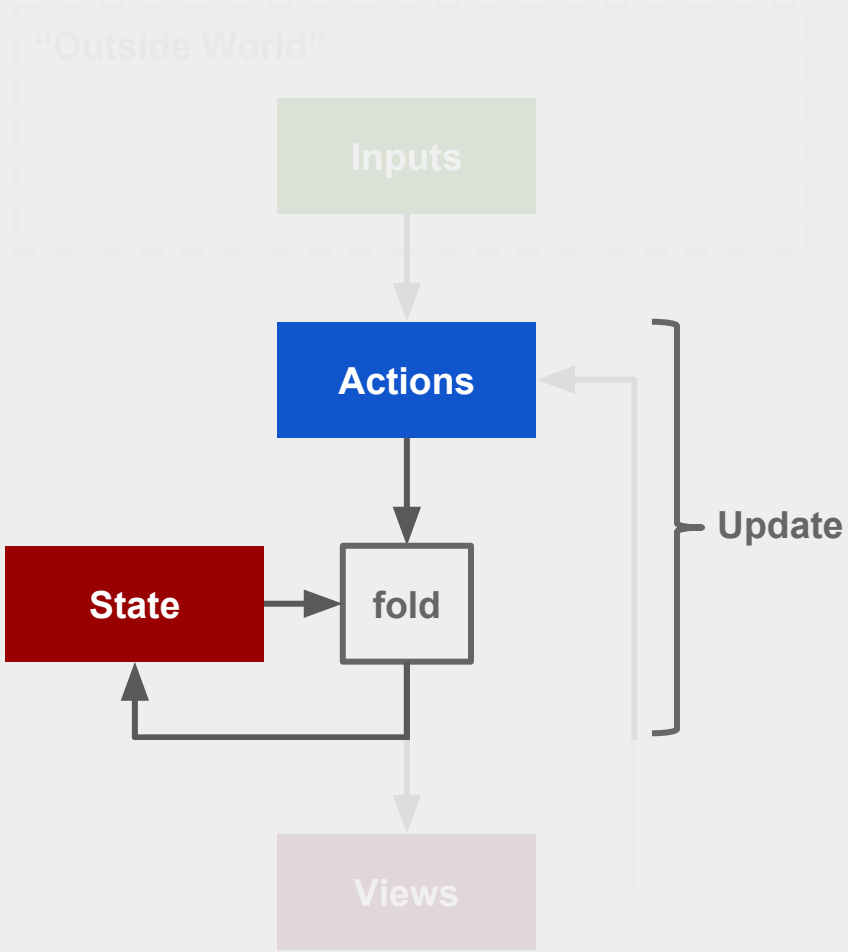
Components

Manageable & Reusable Elm

Image Credit: Gislain Benoit
http://techno-logic-art.com/

“The Elm Architecture”

`StartApp.Simple`



https://github.com/evancz/start-app

Helicopter Parents & Nested State

Parent components

- “hold” the state of their children
- decide when to update the child states
- decide when to render child views

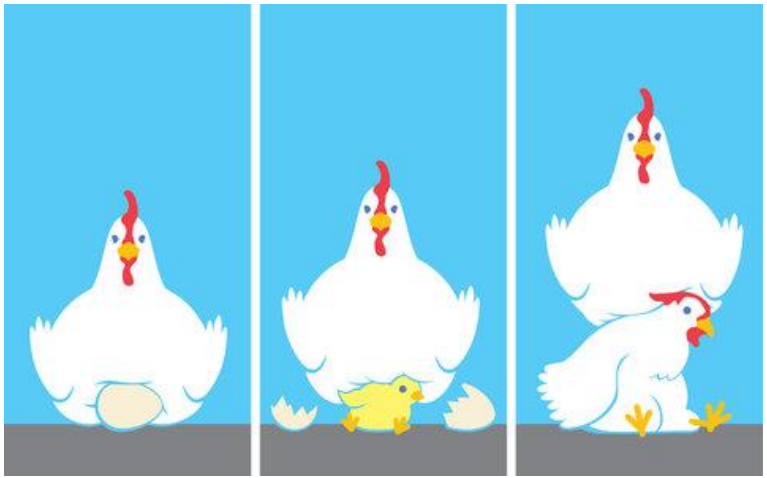


Image Credit: The New York Times
http://learning.blogs.nytimes.com/2013/05/14/do-you-have-helicopter-parents/?_r=0

Helicopter Parents & Nested State

However, it makes sense in terms of modeling:

- A component is *fully* described by its state and its children’s states
- Child states are contextual with respect to the parent
- Maps well to the DOM

Helicopter Parents & Nested State

The result is that

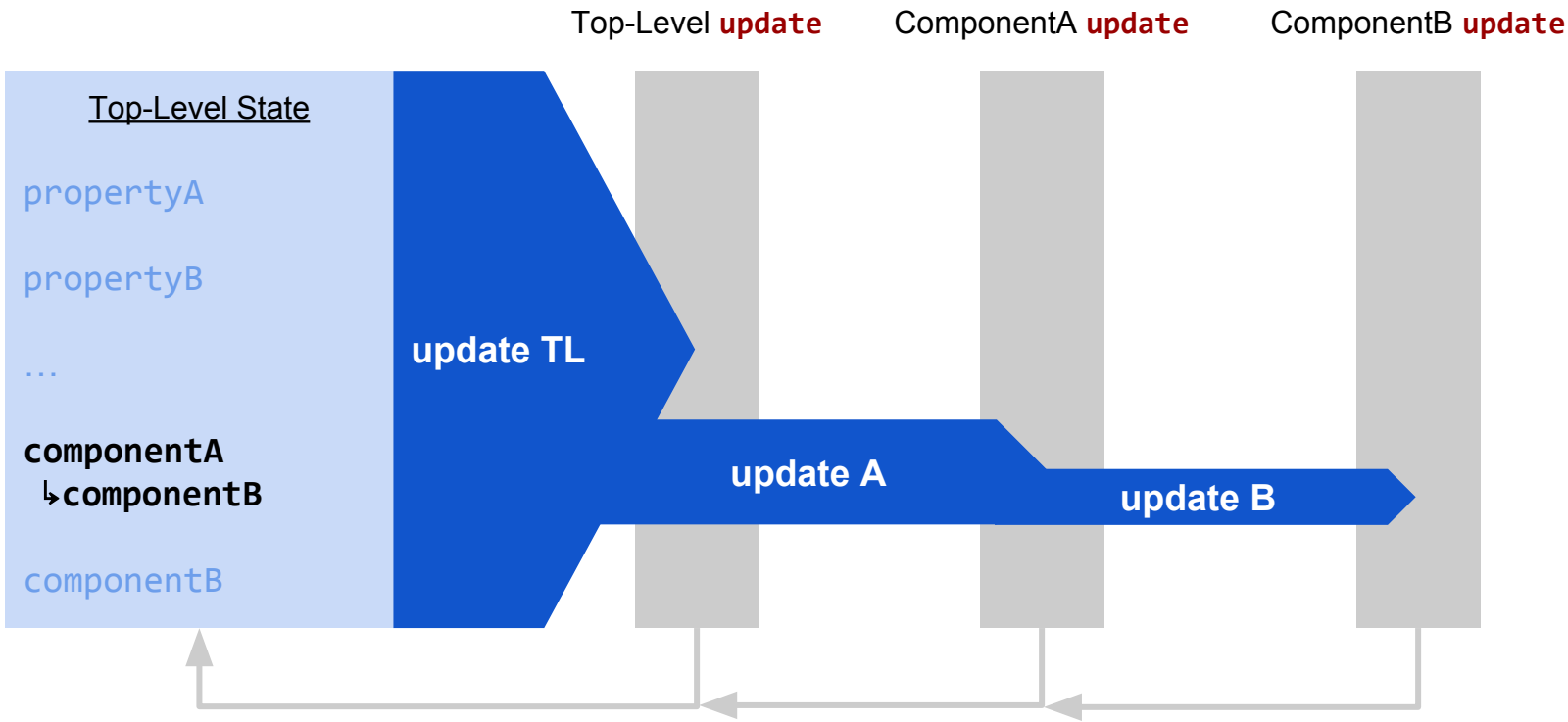
- the top-level application state holds all the state
- the application state is fully described
- you might be thinking: *“global state?!?”*

Image Credit: Sara Berntsson
http://design.tutsplus.com/tutorials/how-to-create-halftone-effects-in-adobe-illustrator--cms-25121

“Sharding” the Application State

- It isn’t really global state but a composite hierarchy of state
- While there is not explicit ownership of data, it is implied through use
- The Elm Architecture “shards” the state into discrete use

“Sharding” the Application State



Let’s Make a Menu

A simple, reusable component.

Making a Menu: Modelling

- List a number of selectable options
- Options are paired to keys of some type
- Think of **a** as some type that must be used consistently

```
7  type alias Model a =
8    { title : String
9    , items : List ( a, String )
10   }
11
12
13 init : String → List ( a, String ) → Model a
14 init title items =
15   { title = title
16   , items = items
17   }
```

Making a Menu: Integrate

- Parent “holds” the model

```
21 model =
22   { width = 320
23   , height = 240
24   , color = red
25   , radius = 50
26   ...
27   , colorMenu = colorMenu
28   }
29
30
31 colorMenu : Menu.Model Color
32 colorMenu =
33   Menu.init
34     "Blob Colour"
35     [ ( Color.red, "Red" )
36     , ( Color.green, "Green" )
37     , ( Color.blue, "Blue" )
38     ]
```

Making a Menu: Integrate

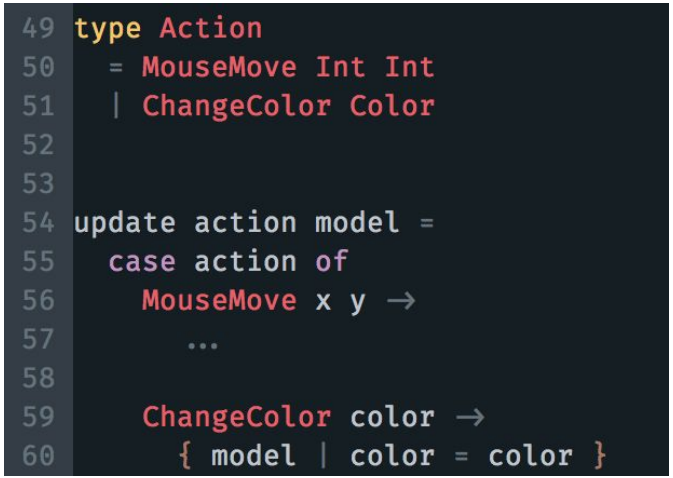
- Parent “holds” the model
- Render the view given the model and the current colour

```
79  view address model =
80    Html.div
81      [ onMouseMove address MouseMove ]
82      [ Html.fromElement
83          (collage
84            model.width
85            model.height
86            (move ( model.px, model.py ) (filled model.co
87              :: List.map viewFood model.food
88            )
89          )
90      , Html.div □ [ Html.text ("Score: " ++ (toString m
91      , viewColorMenu model.color model.colorMenu
92      ]
```

```
101 viewColorMenu : Color → Menu.Model Color → Html.Html
102 viewColorMenu selected menu =
103   Menu.view selected menu
```

Making a Menu: Signals & Actions

- Want the menu to actually *do* something
- Add an action to handle changing colour
- How does the menu communicate this?

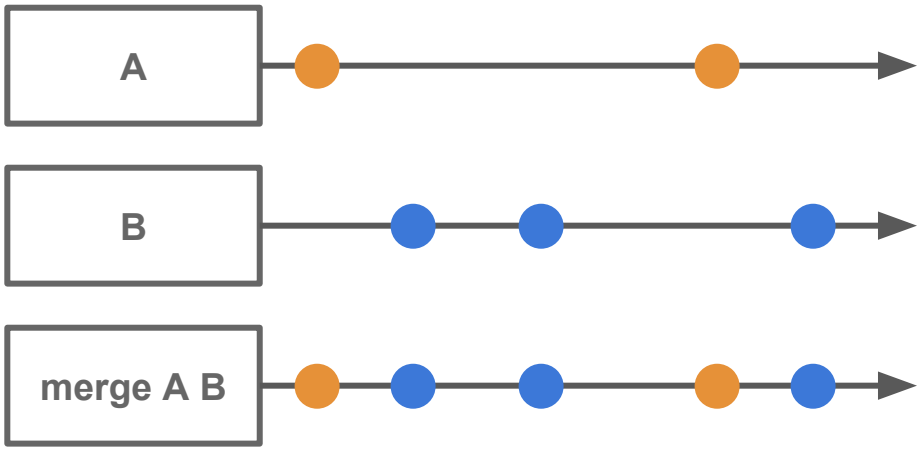


```
49 type Action
50   = MouseMove Int Int
51   | ChangeColor Color
52
53
54 update action model =
55   case action of
56     MouseMove x y →
57       ...
58
59     ChangeColor color →
60       { model | color = color }
```

Making a Menu: Signals & Actions

`Signal`s

- are used for event routing
- propagate **`Action`**s through the Elm architecture to **`Address`**es
- handled like streams (e.g. RxJS)



This is part of what is sometimes referred to as Arrowized Functional Reactive Programming.

Making a Menu: Signals & Actions

- Provide the menu an **`Address`** to send an **`Action`** to when an item is selected
- **`StartApp`** routes this **`Action`** to the application via **`update`**

```elm
type alias Context a =
  { select : Signal.Address a
  }
```

```elm
itemView : a -> Context a -> ( a, String ) -> Html.Html
itemView selection context item =
  let
    fontWeight =
      if selection == (fst item) then
        "bold"
      else
        "normal"
  in
    Html.div
      [ Attr.style [ ( "fontWeight", fontWeight ) ]
      , onClick context.select (fst item)
      ]
      [ Html.text (snd item) ]
```

Making a Menu: Signals & Actions

- `Signal.forwardTo` creates a new `Address` forwarding to the given one, tagging it with an `Action`
- Messages sent to the select `Address` will now arrive in Main's `update` as the action `ChangeColor`
- Note the action type is `ChangeColor Color` and the `Context`'s `select` is `Address Color`

```
93  viewColorMenu : Signal.Address Action → Color → Menu.Model Color → Html.Html
94  viewColorMenu address selected menu =
95    let
96      context =
97        { select = Signal.forwardTo address ChangeColor
98        }
99    in
100     Menu.view selected context menu
```

Making a Menu: Actions on the Menu

- The menu can trigger **`Action`**s on itself
- We define a couple **`Action`**s on the menu and add an **`update`** function
- The **`Context`** must now contain an **`Address`** to which these **`Action`**s are sent

```
29 type Action
30   = Collapse
31   | Expand
32
33
34 update : Action -> Model a -> Model a
35 update action model =
36   case action of
37     Collapse ->
38       { model | expanded = False }
39
40     Expand ->
41       { model | expanded = True }
```

```
15 type alias Context a =
16   { select : Signal.Address a
17   , action : Signal.Address Action
18   }
```

Making a Menu: Actions on the Menu

- The main application must provide the **`Address`** and **`update`** the menu
- It also needs to tag the **`Address`** with its own **`Action`** so we know how to handle it
- This is a concrete example of the “sharding” mentioned before

```
49 type Action
50   = MouseMove Int Int
51   | ChangeColor Color
52   | ModifyColorMenu Menu.Action
53
54
55 update action model =
56   case action of
57     MouseMove x y ->
58       ...
59
60     ChangeColor color ->
61       { model | color = color }
62
63     ModifyColorMenu menuAction ->
64       { model
65         | colorMenu = Menu.update menuAction model.colorMenu
66       }
```

Making a Menu: Actions on the Menu

- The main application must provide the **`Address`** and **`update`** the menu
- It also needs to tag the **`Address`** with its own **`Action`** so we know how to handle it
- This is a concrete example of the “sharding” mentioned before

```
110 viewColorMenu address selected menu =
111   let
112     context =
113       { select = Signal.forwardTo address ChangeColor
114       , action = Signal.forwardTo address ModifyColorMenu
115       }
116   in
117     Menu.view selected context menu
```

Making a Menu: Reuse!

- With all the work in place up to now, reuse is pretty simple
- Add another model to the top level model, this time a radius menu
- Add **`Action`**s to handle changes in radius and to the radius menu itself

```
radiusMenu : Menu.Model Float
radiusMenu =
  Menu.init
    "Blob Size"
    [ ( 10, "Tiny" )
    , ( 25, "Small" )
    , ( 50, "Normal" )
    , ( 99, "Seriously?" )
    ]


type Action
  = MouseMove Int Int
  | ChangeColor Color
  | ChangeRadius Float
  | ModifyColorMenu Menu.Action
  | ModifyRadiusMenu Menu.Action
```

Making a Menu: Reuse!

- Cover the new **`Action`** cases in the **`update`** function

```elm
update action model =
  case action of
    MouseMove x y ->
      ...

    ChangeColor color ->
      { model | color = color }

    ChangeRadius radius ->
      { model | radius = radius }

    ModifyColorMenu menuAction ->
      { model | colorMenu = Menu.update menuAction model.colorMenu }

    ModifyRadiusMenu menuAction ->
      { model | radiusMenu = Menu.update menuAction model.radiusMenu }
```

Making a Menu: Reuse!

- Render the menu exactly the same way
- Do absolutely nothing code-wise to the menu component!

```
129 viewRadiusMenu : Signal.Address Action → Float → Menu.Model Float → Html.Html
130 viewRadiusMenu address selected menu =
131   let
132     context =
133       { select = Signal.forwardTo address ChangeRadius
134       , action = Signal.forwardTo address ModifyRadiusMenu
135       }
136   in
137     Menu.view selected context menu
```

- DRY up your code because you aren’t a savage

Thanks!

Comments & Questions