Tame the frontend with Elm

Jeremy Fairbank / @elpapapapollo
Software is broken.
We are here to fix it.
Say hello@testdouble.com
Happiness
✓ Easier to write code
✓ Easier to refactor
✓ Easier to write tests
No runtime exceptions in practice.
No undefined is not a function
Fast

[Bar chart showing speed in milliseconds for different frameworks and libraries, with the text "Shorter bars are better." and a legend indicating "Naive" and "Optimized" bars.]
One framework. No fatigue.
Functional
greet name =
  "Hello, " ++ name

++
greet "Fluent Conf"

-- Hello, Fluent Conf
greet name = "Hello, " ++ name

greet "Fluent Conf"

-- Hello, Fluent Conf
greet name = 
  "Hello, " ++ name

greet "Fluent Conf"

-- Hello, Fluent Conf
greet name = "Hello, " ++ name

Single Expression

greet "Fluent Conf"

-- Hello, Fluent Conf
greet name = "Hello, " ++ name

greet "Fluent Conf"

-- Hello, Fluent Conf
greet name =
"Hello, " ++ name

greet "Fluent Conf"

-- Hello, Fluent Conf
Pure

Data in

Data out
Pure  No side effects
Pure Predictable and Testable!
add \ x \ y \ = \ x \ + \ y

add\ 2\ 3\ \ ==\ 5
add\ 2\ 3\ \ ==\ 5
add\ 2\ 3\ \ ==\ 5
add \( x \ y \ = \ x + y \)
add \ x \ y \ =
\ x + y

\rightarrow \ add \ 2 \ 3 \ == \ 5
\rightarrow \ add \ 2 \ 3 \ == \ 5
\rightarrow \ add \ 2 \ 3 \ == \ 5
Expressive
Terse and declarative code
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list = List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list = List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list = List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list = List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list = List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
myList = [1, 2, 3, 4, 5]

double n = n * 2

doubleNumbers list =
    List.map double list

doubleNumbers myList -- [2, 4, 6, 8, 10]
Curried Functions
Building blocks
\[ \text{add } x \ y = x + y \]

\[ \text{add } 1 \ 2 \quad -- \quad 3 \]

\[ (\text{add } 1) \ 2 \quad -- \quad 3 \]
\[ \text{add } x \ y = x + y \]

\[ \rightarrow \ \text{add } 1 \ 2 \quad -- \quad 3 \]

\[ (\text{add } 1) \ 2 \quad -- \quad 3 \]
add x y = x + y

add 1 2 -- 3

→ (add 1) 2 -- 3
add \( x \ y = x + y \)

add 1 2 \(--\) 3

New function created

\((\text{add} \ 1) \ 2 \ -- \ 3\)
increment = add 1

increment 2   --  3
increment 41  --  42
increment = add 1

increment 2 -- 3
increment 41 -- 42
increment = add 1

increment 2  --  3
increment 41  --  42
Pipes
Expressive Chaining
greet name = "Hello, " ++ name
exclaim phrase = phrase ++ "!"

exclaim (greet (String.toUpperCase "Tucker"))
"Tucker"

|> String.toUpper
|> greet
|> exclaim
"Tucker"

\[\text{String.toUpper} \rightarrow \text{greet} \rightarrow \text{exclaim}\]
|> String.toUpperCase "Tucker"
|> greet
|> exclaim
"TUCKER"

|> greet
|> exclaim
|> greet "TUCKER"
|> exclaim
"Hello, TUCKER"

|> exclaim
 |> exclaim "Hello, TUCKER"
"Hello, TUCKER!"
No Runtime Exceptions
Strong Static Types

```plaintext
life : Int
life = 42

greeting : String
greeting = "Hello World"

isTrue : Bool
isTrue = True

numbers : List Int
numbers = [1, 2, 3]
```
Strong Static Types

life : Int
life = 42

greeting : String
greeting = "Hello World"

isTrue : Bool
isTrue = True

numbers : List Int
numbers = [1, 2, 3]
Strong Static Types

```plaintext
life : Int
life = 42

greeting : String
greeting = "Hello World"

isTrue : Bool
isTrue = True

numbers : List Int
numbers = [1, 2, 3]
```
greet : String -> String

\[
greet \ name = \ "Hello, \ " \ ++ \ name
\]

add : Int -> Int -> Int

\[
add \ x \ y = \\
x + y
\]
greet : String -> String

```haskell
greet name =
  "Hello, " ++ name
```

add : Int -> Int -> Int

```haskell
add x y =
  x + y
```
greet : String -> String

greet name =
  "Hello, " ++ name

add : Int -> Int -> Int

add x y =
  x + y
greet : String -> String

```haskell
greet name =
  "Hello, " ++ name
```

add : Int -> (Int -> Int)

```haskell
add x y =
  x + y
```
The argument to function `greet` is causing a mismatch.

11|     greet 42
    ^^^
Function `greet` is expecting the argument to be:

    String

But it is:

    number
Immutable Data
Safe and Consistent
Records

dog : { name : String, age : Int }
dog =
   { name = "Tucker",
     age = 11
   }

dog.name -- "Tucker"
dog.age -- 11
Records

dog : { name : String, age : Int }
dog =

    { name = "Tucker"
    , age = 11
    }

dog.name -- "Tucker"
dog.age -- 11
Records

dog : { name : String, age : Int } 
dog =
    { name = "Tucker"
    , age = 11
    }

dog.name -- "Tucker"
dog.age -- 11
Records

dog : { name : String, age : Int }
dog =
    { name = "Tucker",
      age = 11
    }

dog.name -- "Tucker"
dog.age -- 11
Create New Data

```plaintext
haveBirthday dog =
   { dog | age = dog.age + 1 }

dog =  { name = "Tucker", age = 11 }
olderDog = haveBirthday dog

olderDog.age  --  12
olderDog.name  --  "Tucker"

dog.age  --  11
dog.name  --  "Tucker"
```
Create New Data

haveBirthday dog =
{  dog  |  age = dog.age + 1  }

dog =  {  name = "Tucker", age = 11  }
olderDog = haveBirthday dog

olderDog.age  --  12
olderDog.name  --  "Tucker"

dog.age  --  11
dog.name  --  "Tucker"
Create New Data

```python
haveBirthday dog = 
   { dog | age = dog.age + 1 }

dog = { name = "Tucker", age = 11 }
olderDog = haveBirthday dog

olderDog.age -- 12
olderDog.name -- "Tucker"

dog.age -- 11
dog.name -- "Tucker"
```
Create New Data

```haskell
haveBirthday dog =
    { dog | age = dog.age + 1 }

dog =  { name = "Tucker", age = 11 }
olderDog = haveBirthday dog

olderDog.age -- 12
olderDog.name -- "Tucker"

dog.age -- 11
dog.name -- "Tucker"
```
Create New Data

```javascript
haveBirthday dog =
    { dog | age = dog.age + 1 };

dog = { name = "Tucker", age = 11 };
olderDog = haveBirthday dog;

olderDog.age -- 12
olderDog.name -- "Tucker"

dog.age -- 11
dog.name -- "Tucker"
```
Create New Data

```javascript
haveBirthday dog =
    { dog | age = dog.age + 1 }

dog = { name = "Tucker", age = 11 }
olderDog = haveBirthday dog

olderDog.age -- 12
olderDog.name -- "Tucker"

dog.age -- 11
dog.name -- "Tucker"
```
Custom Types

Domain-specific code
type alias Dog =
{
  name : String,
  age : Int,
  breed : Breed
}

type Breed
= Sheltie
  | Poodle
type alias Dog =
{
  name : String,
  age : Int,
  breed : Breed
}

type Breed
  = Sheltie
  | Poodle
type alias Dog =
{
  name : String,
  age : Int,
  breed : Breed
}

type Breed
  = Sheltie
  | Poodle
```python
dog : Dog
dog =
{
    name = "Tucker",
    age = 11,
    breed = Sheltie
}
```
```
dog : Dog
dog =
{
    name = "Tucker",
    age = 11,
    breed = Sheltie
}
```
```python
dog : Dog
dog =
{
    name = "Tucker",
    age = 11,
    breed = Sheltie
}
```
No null or undefined
type Maybe a = Just a | Nothing
type Maybe a = Just a | Nothing

Wraps the successful value
type Maybe a = Just a | Nothing

Represents no result or missing value
type Maybe a = Just a | Nothing
Either I have Just the value a, or I have Nothing.
divide : Float -> Float -> Maybe Float
divide x y =
    if y == 0 then
        Nothing
    else
        Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
divide : Float -> Float -> Maybe Float

divide x y =
    if y == 0 then
        Nothing
    else
        Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
divide : Float -> Float -> Maybe Float
divide x y =
  if y == 0 then Nothing
  else Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
divide : Float -> Float -> Maybe Float
divide x y =
  if y == 0 then
    Nothing
  else
    Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
divide : Float -> Float -> Maybe Float
divide x y =
    if y == 0 then
        Nothing
    else
        Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
divide :: Float -> Float -> Maybe Float
divide x y =
  if y == 0 then
    Nothing
  else
    Just (x / y)

divide 4 2 -- Just 2
divide 4 0 -- Nothing
case divide 4 2 of
  Just n ->
    "Result is " ++ (toString n)

  Nothing ->
    "No Result"
case divide 4 2 of
  Just n ->
  "Result is " ++ (toString n)

Nothing ->
  "No Result"
case divide 4 2 of
  Just n ->
    "Result is " ++ (toString n)
  Nothing ->
    "No Result"
case divide 4 2 of
    Just n ->
        "Result is " ++ (toString n)
    Nothing ->
        "No Result"
case divide 4 2 of
    Just n ->
        "Result is " ++ (toString n)
    Nothing ->
        "No Result"
Exhaustive matching

This `case` does not have branches for all possibilities.

```
21|> case divide 4 2 of
22|>   Just n ->
23|>     "Result is " ++ (toString n)
```

You need to account for the following values:

Maybe.Nothing

Add a branch to cover this pattern!
The Elm Architecture
Demos

The Elm Architecture in Action
Getting Started

- elm-lang.org
- elm-lang.org/examples
- guide.elm-lang.org
- www.elm-tutorial.org
- builtwithelm.co
- Slack
- elmlang.herokuapp.com
Thank you!

Jeremy Fairbank / @elpapapapollo

Say hello@testdouble.com