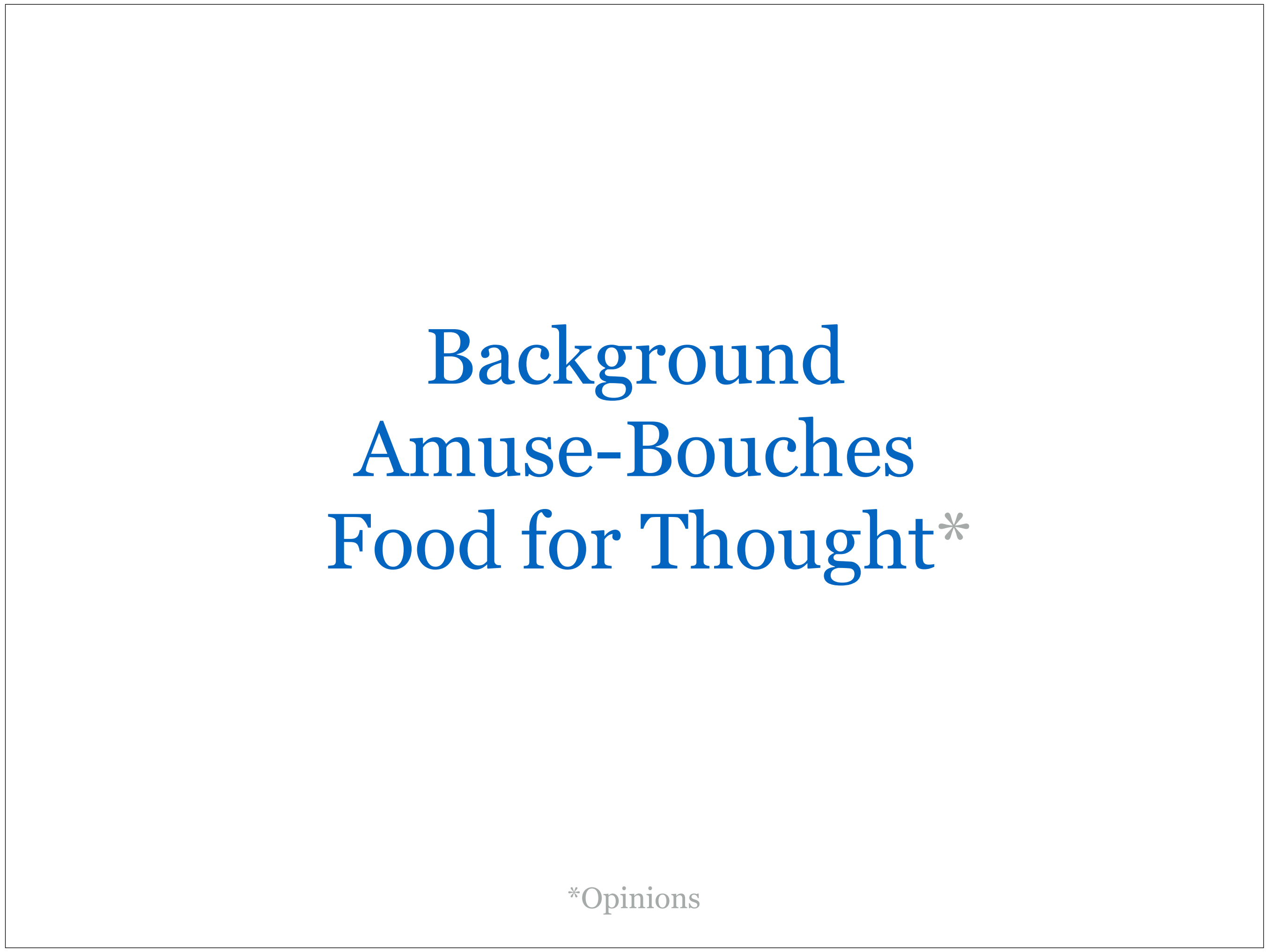
# Functional Programming *for* Fun & Profit*

*Or: How I Learned to Stop Worrying and Love Shipping Haskell Code*

Daniel Gąsienica
March 17, 2017

*after accomplishing step 2: ???

Hello everyone! Thanks for joining me today for brownbag called ‘Functional Programming for Fun and Profit — Or: How I Learned to Stop Worrying and Love Shipping Haskell Code’.

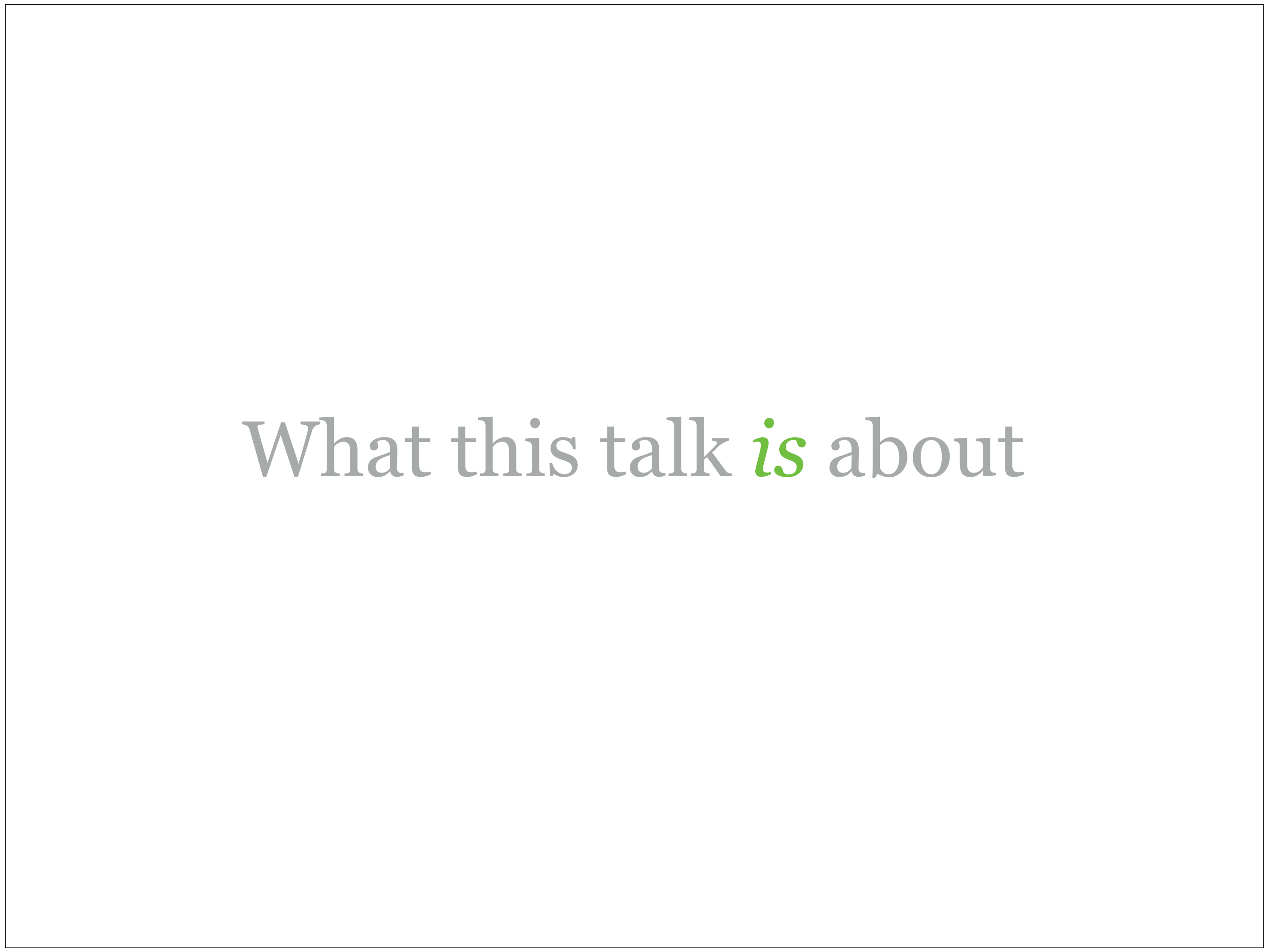


Background
Amuse-Bouches
Food for Thought*

*Opinions

This talk is broken down into three acts:

First, I’ll talk about how I got into functional programming.
Second, I will present you a few of my favorite concepts that I learned.
Third, I will share some food for thought I have found on my journey so far.



# What this talk *is* about

This talk is about sharing the excitement of functional programming by giving you some ideas what it’s good for and how it addresses problems we have in imperative programming.

It is about picking a few concepts and diving into them.

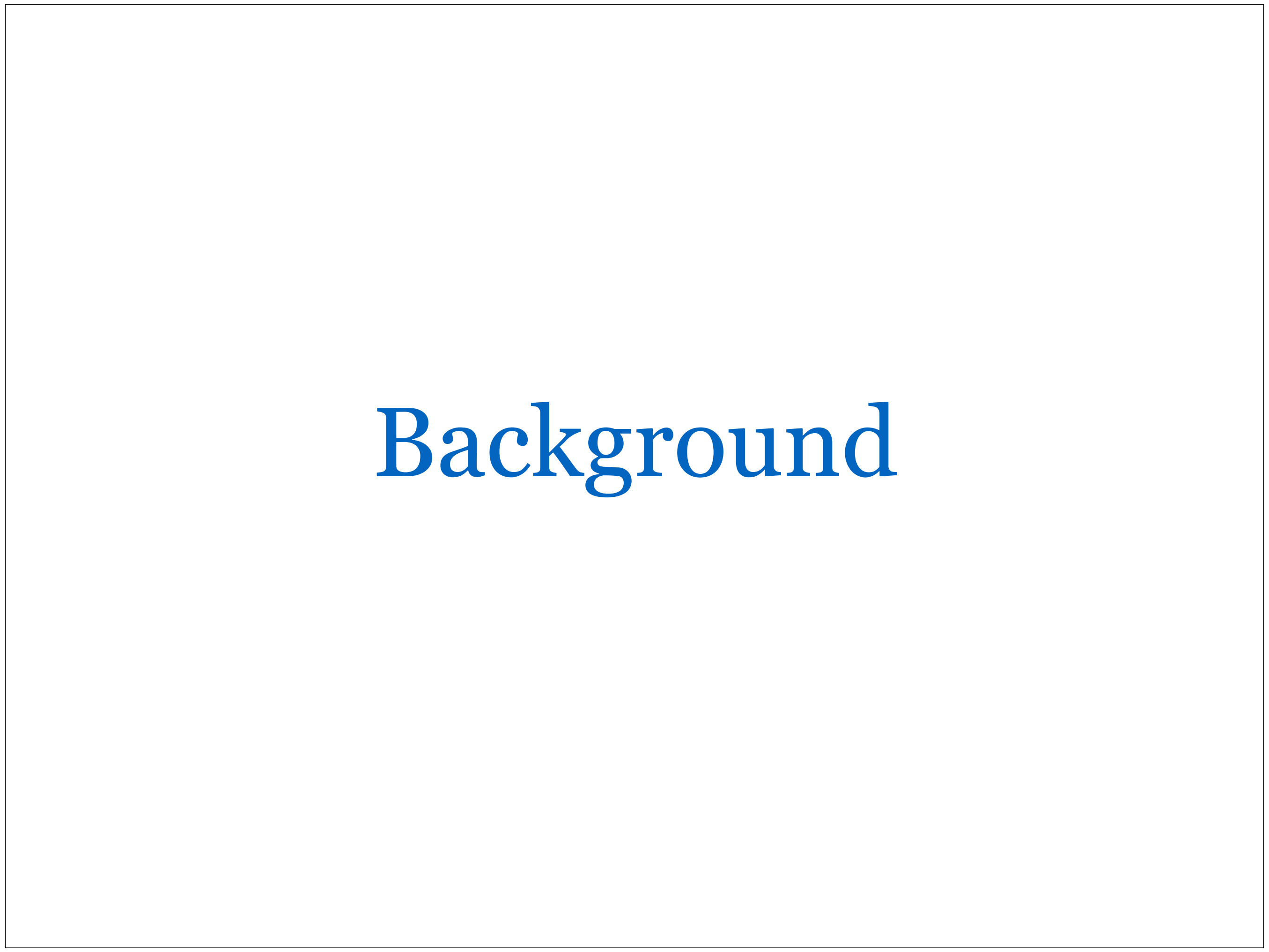
It is supposed to teach you a few things, but more importantly, it is intended to spark your curiosity and to question the status quo.

# What this talk *is not* about

This talk is not about teaching you the syntax of a new language or explain every single concept touched upon in great depth. Unfortunately, the time for that is simply too short.

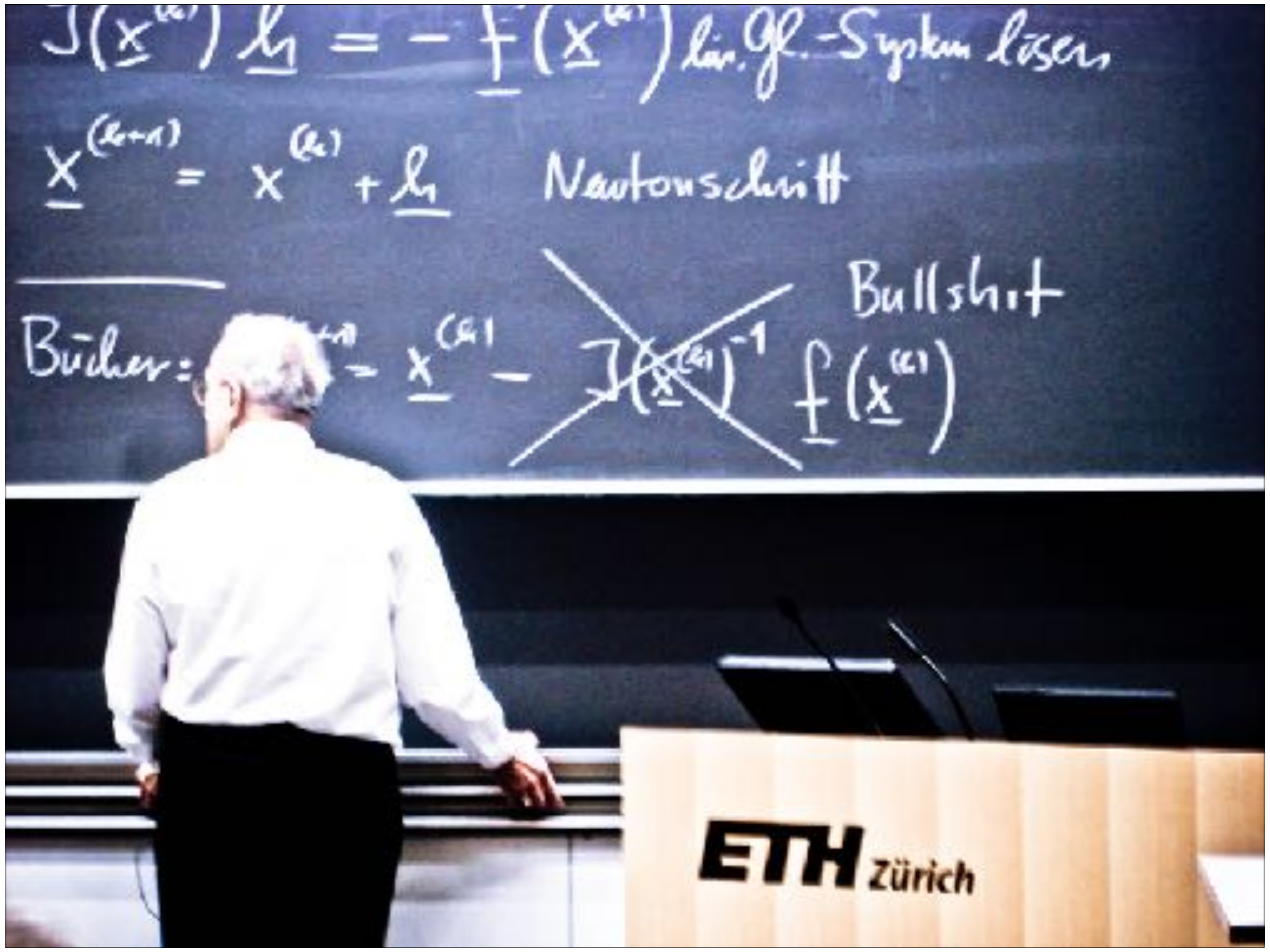
However, I love this stuff, so if anything is unclear, please speak up and I’ll try my best to answer your question.

If your question needs more time, we can continue the conversation afterwards.



# Background

Before I dive into the material, I wanted to briefly go back in time and explain where my interest in functional programming comes from.


$J(\underline{x}^{(k)})\,\underline{h} = -\underline{f}(\underline{x}^{(k)})$ lin. Gl.-System lösen
$\underline{x}^{(k+1)} = x^{(k)} + \underline{h}$ Newtonschritt
Bücher: $\underline{x}^{(k+1)} = \underline{x}^{(k)} - \cancel{J(\underline{x}^{(k)})^{-1}}\,\underline{f}(\underline{x}^{(k)})$
Bullshit
ETH Zürich

In 2008, I attended a class called ‘Formal Methods & Functional Programming’ at my university in Switzerland. This is where I was first exposed to—surprise…

…a language called *Haskell.* I was immediately fascinated by how concise, elegant, and… how *different* it was.

However, this talk is *not* about Haskell. It’s more about the lessons it taught me and that can also be learned from other statically typed (and some pure) functional programming languages such as ML, PureScript, Elm, etc.

# Thanksgiving 2015

Photo: https://commons.wikimedia.org/wiki/User:Braniff747SP

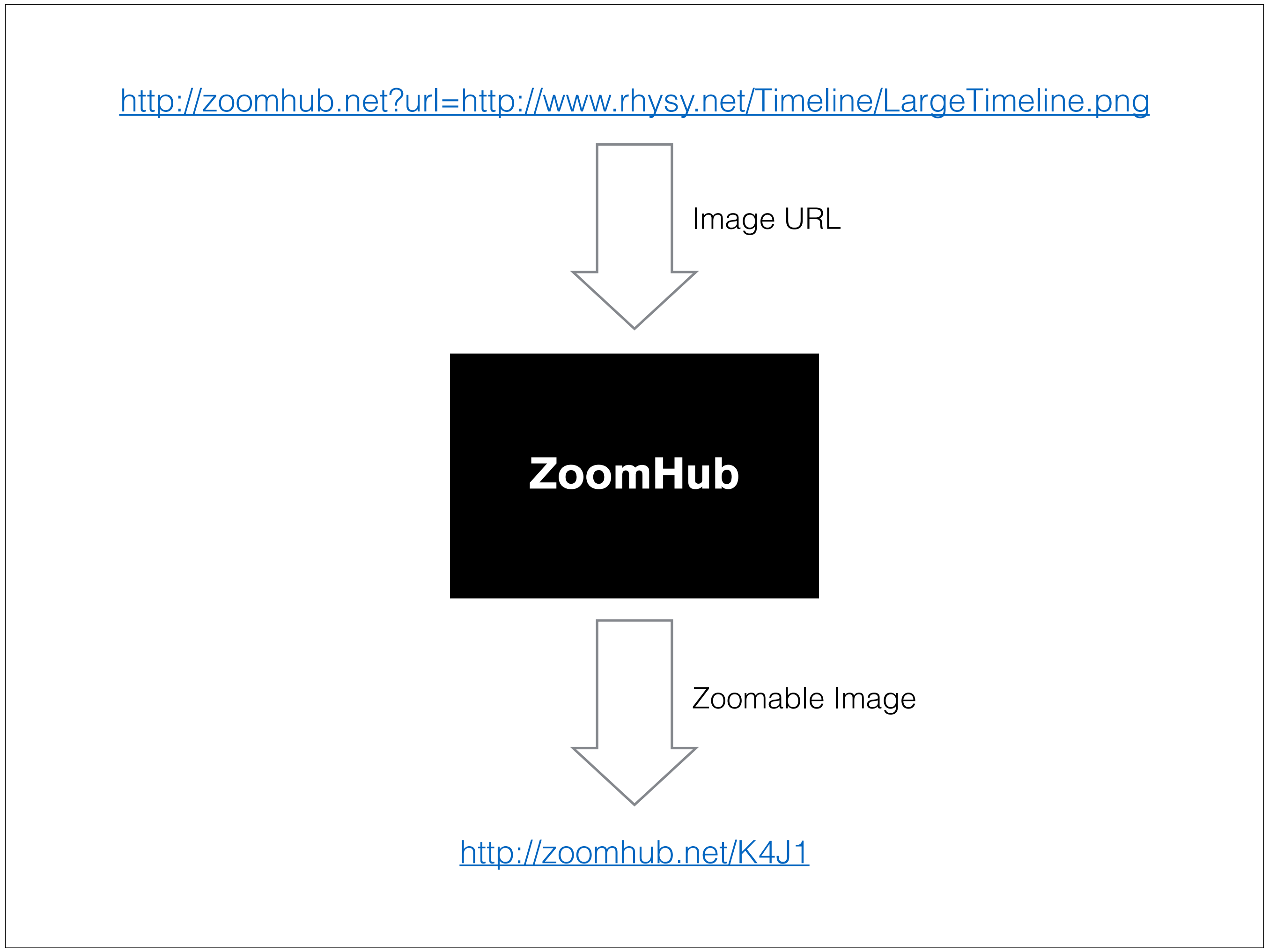
After a few failed attempts to learn Haskell since that class I took in 2008, I was riding the train from California to Seattle after Thanksgiving and had 31+ hours to kill. That’s when I decided to finally tackle learning Haskell.

Instead of programming with toy examples, I chose to learn by writing real-world code.

How? I started porting a Node.js web service that Aseem, myself, and a few of our friends have built.

zoomhub

That web service is called ZoomHub and has been running in production on Haskell since April of last year (2016).



http://zoomhub.net?url=http://www.rhysy.net/Timeline/LargeTimeline.png

Image URL

ZoomHub

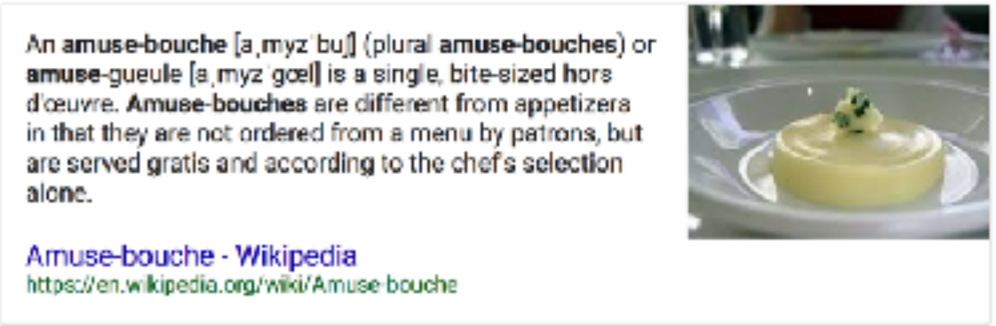
Zoomable Image

http://zoomhub.net/K4J1

[DEMO]

Universe timeline: http://zoomhub.net/K4J1

# Amuse-Bouches

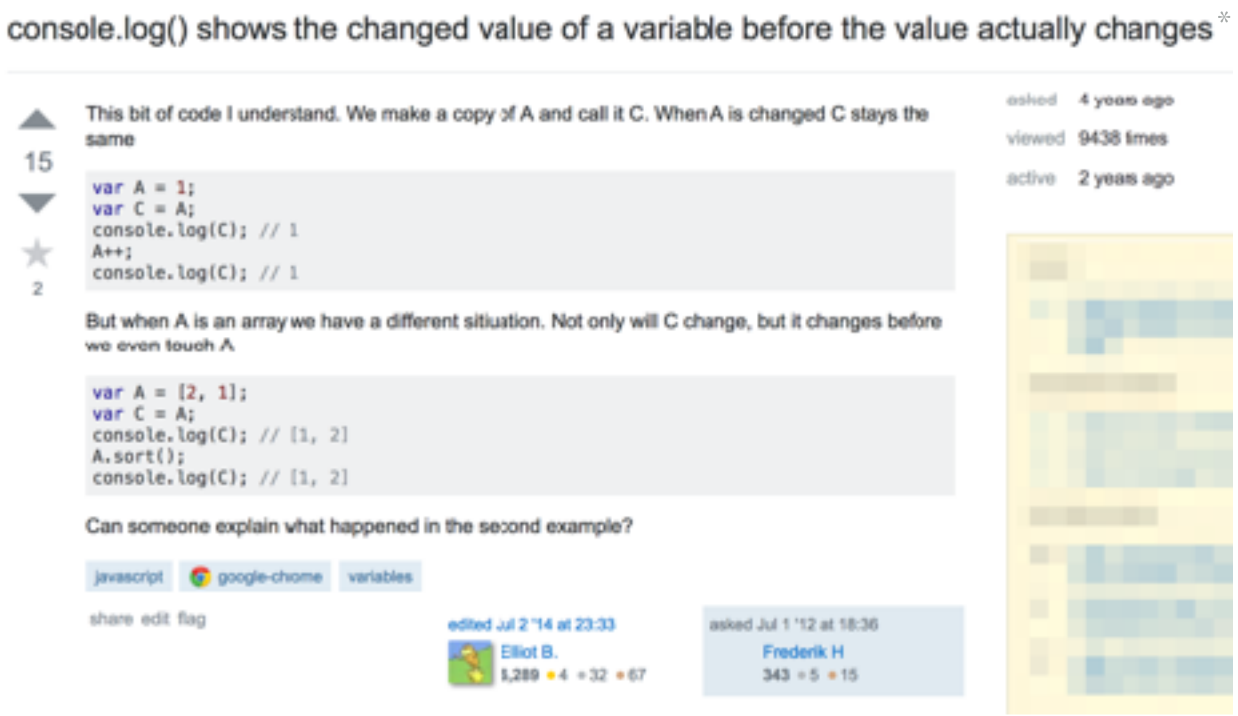


An **amuse-bouche** [aˌmyzˈbuʃ] (plural **amuse-bouches**) or **amuse**-gueule [aˌmyzˈɡœl] is a single, bite-sized hors d'œuvre. **Amuse-bouches** are different from appetizers in that they are not ordered from a menu by patrons, but are served gratis and according to the chef's selection alone.

Amuse-bouche - **Wikipedia**
https://en.wikipedia.org/wiki/Amuse-bouche

# Immutability & The Value of Values

# The Pain



## console.log() shows the changed value of a variable before the value actually changes *

15

2

This bit of code I understand. We make a copy of A and call it C. When A is changed C stays the same

```
var A = 1;
var C = A;
console.log(C); // 1
A++;
console.log(C); // 1
```

But when A is an array we have a different sitiuation. Not only will C change, but it changes before we even touch A

```
var A = [2, 1];
var C = A;
console.log(C); // [1, 2]
A.sort();
console.log(C); // [1, 2]
```

Can someone explain what happened in the second example?

edited Jul 2 '14 at 23:33
Elliot B.
5,289 ●4 ●32 ●67

asked Jul 1 '12 at 18:36
Frederik H
343 ●5 ●15

asked 4 years ago

viewed 9438 times

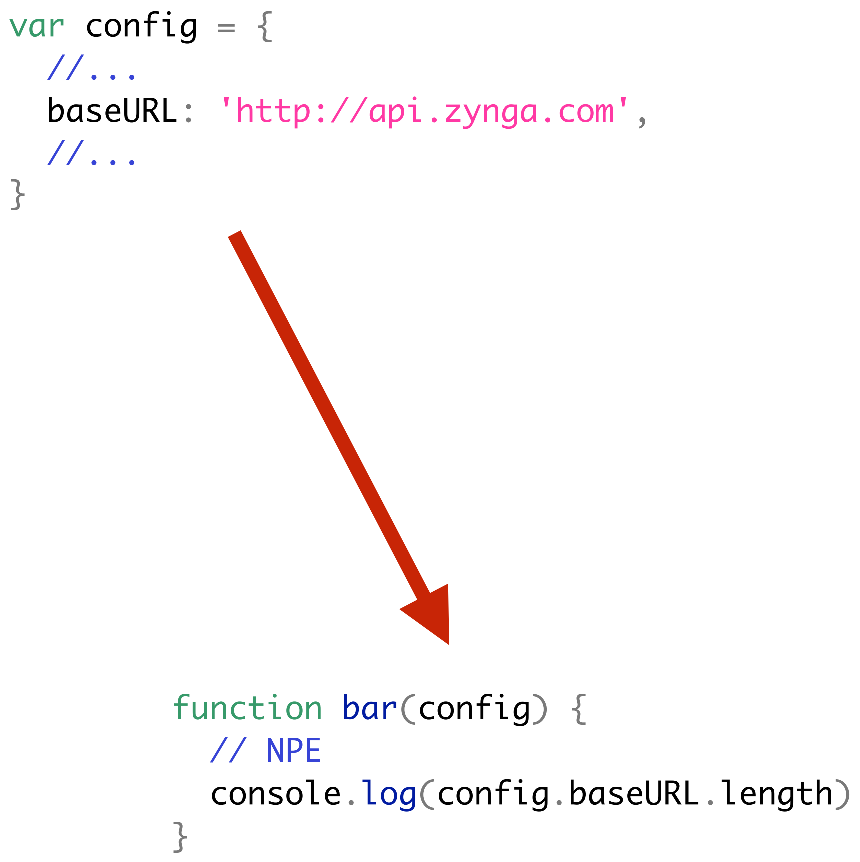
active 2 years ago

* Fixed in recent versions of WebKit/Chrome

# The Bugs

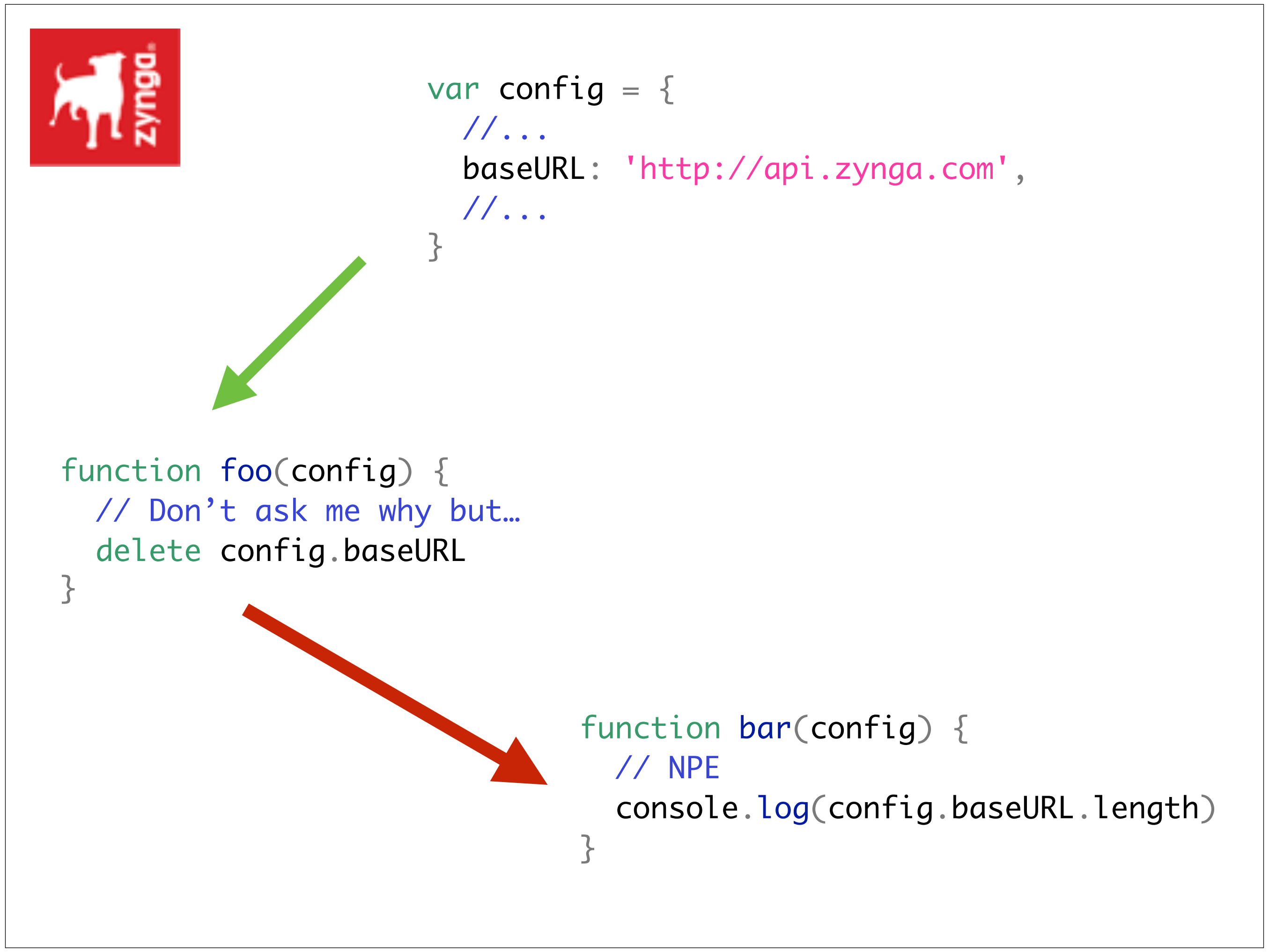
```
var config = {
  //...
  baseURL: 'http://api.zynga.com',
  //...
}
```

```
function bar(config) {
  console.log(config.baseURL.length)
}
```



```
var config = {
  //...
  baseURL: 'http://api.zynga.com',
  //...
}
```

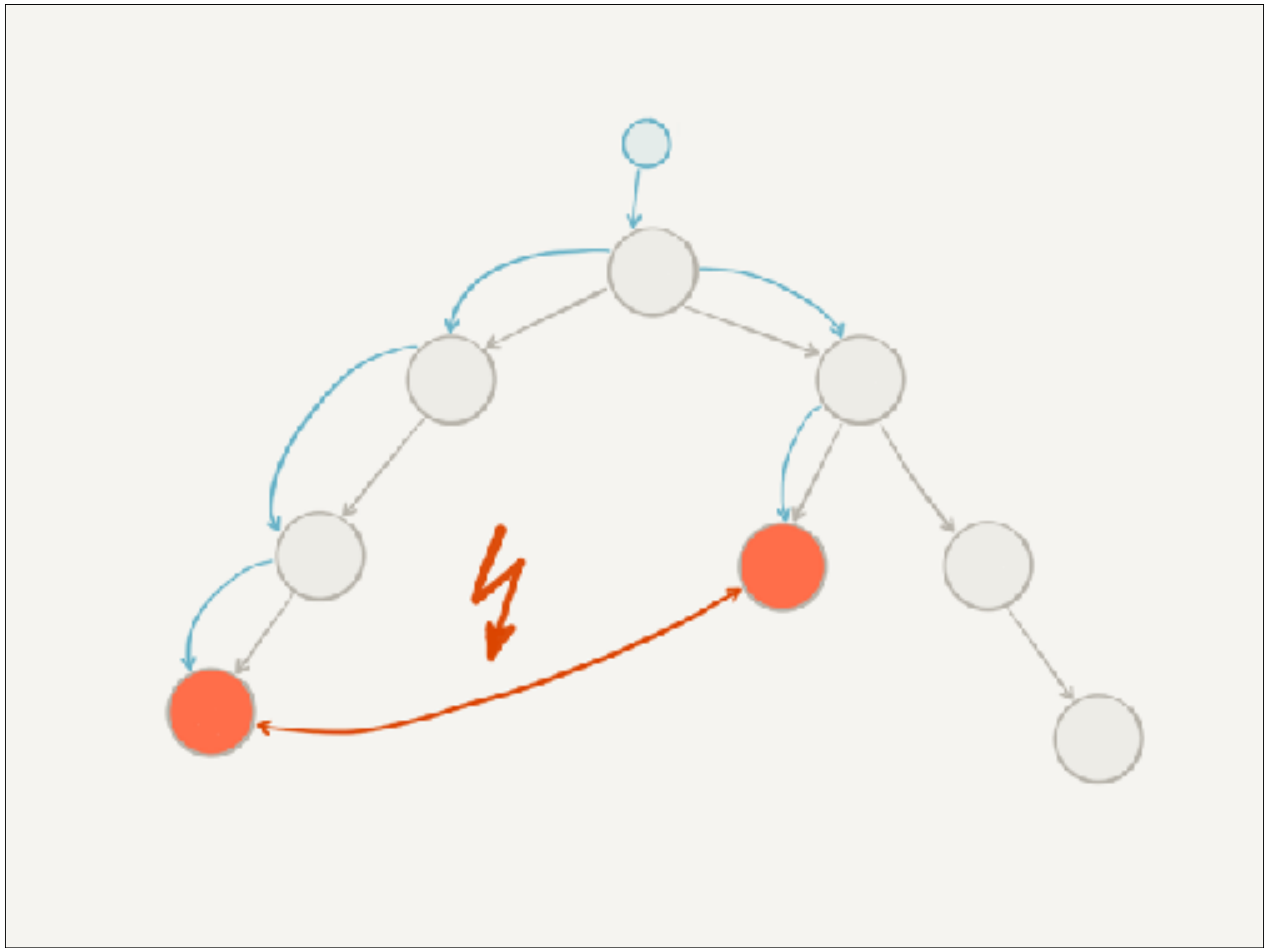
```
function bar(config) {
  // NPE
  console.log(config.baseURL.length)
}
```



```
var config = {
  //...
  baseURL: 'http://api.zynga.com',
  //...
}
```

```
function foo(config) {
  // Don’t ask me why but…
  delete config.baseURL
}
```

```
function bar(config) {
  // NPE
  console.log(config.baseURL.length)
}
```



# The Confusion

```
> 1 === 1
true

> true === true
true

> "hello" === "hello"
true
```

```
> 1 === 1
true

> true === true
true

> "hello" === "hello"
true
```

```
> [] === []
false

> [1, 2] === [1, 2]
false

> {} === {}
false

> {"a": "b"} === {"a": "b"}
false
```

```
> 1 === 1
true

> true === true
true

> "hello" === "hello"
true

> [] === []
false

> [1, 2] === [1, 2]
false

> {} === {}
false

> {"a": "b"} === {"a": "b"}
false
```

```
> 1 == 1
True

> True == True
True

> "hello" == "hello"
True

> [] == []
True

> [1, 2] == [1, 2]
True

> Map.fromList [] == Map.fromList []
True

> Map.fromList [("a", "b")] == Map.fromList [("a", "b")]
True
```

```
> let a = [3, 1, 2]

> let b = a.sort()

> b
[1, 2, 3]
```

```
> let a = [3, 1, 2]

> let b = a.sort()

> b
[1, 2, 3]

> a
[1, 2, 3]
```

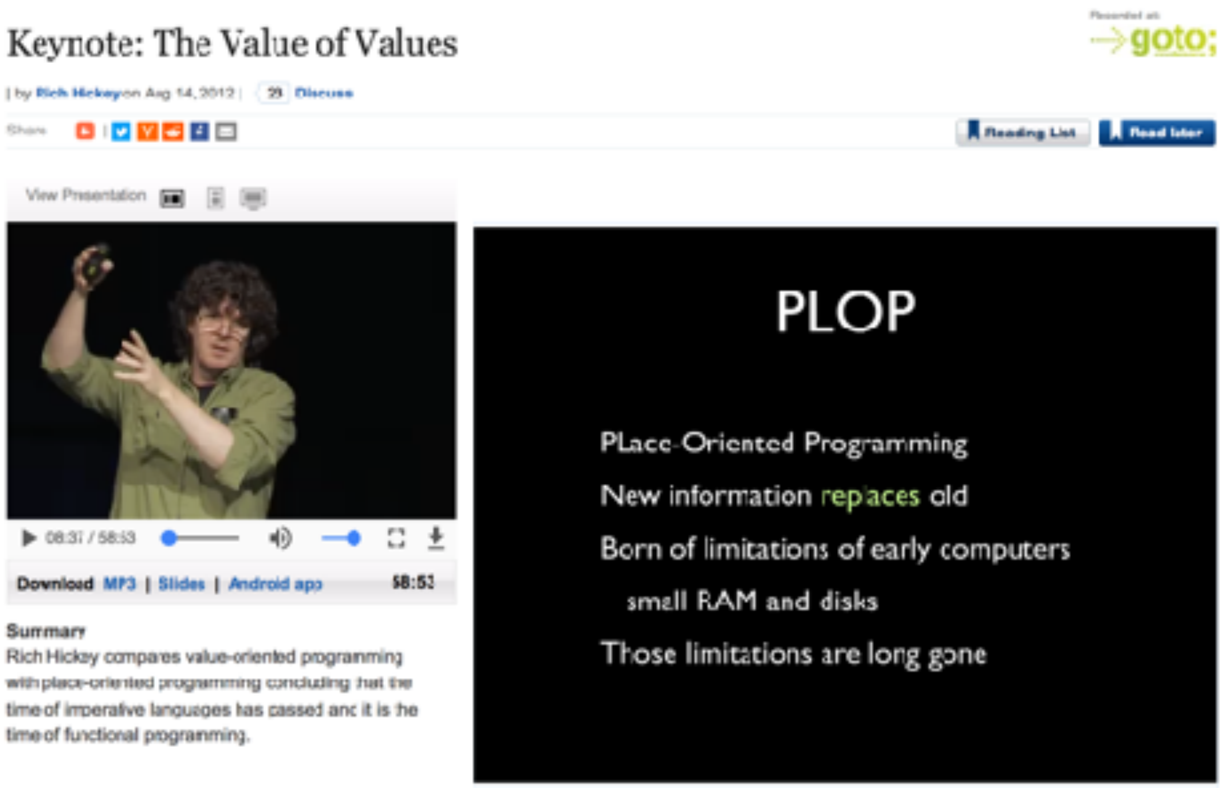
```
> let a = [3, 1, 2]

> let b = a.sort()

> b
[1, 2, 3]

> a
[1, 2, 3]
```

```
> let a = [3, 1, 2]

> let b = sort a

> b
[1, 2, 3]

> a
[3, 1, 2]
```

- space = function
-          application
- i.e. JavaScript: sort(a)

# Conclusion

Abandon distinction between values and references and treat everything as immutable values.



Keynote: The Value of Values

| by Rich Hickey on Aug 14, 2012 | 29 Discuss

Summary

Rich Hickey compares value-oriented programming with place-oriented programming concluding that the time of imperative languages has passed and it is the time of functional programming.

PLOP

PLace-Oriented Programming

New information replaces old

Born of limitations of early computers

small RAM and disks

Those limitations are long gone

https://www.infoq.com/presentations/Value-Values

# null

The Billion-Dollar Mistake

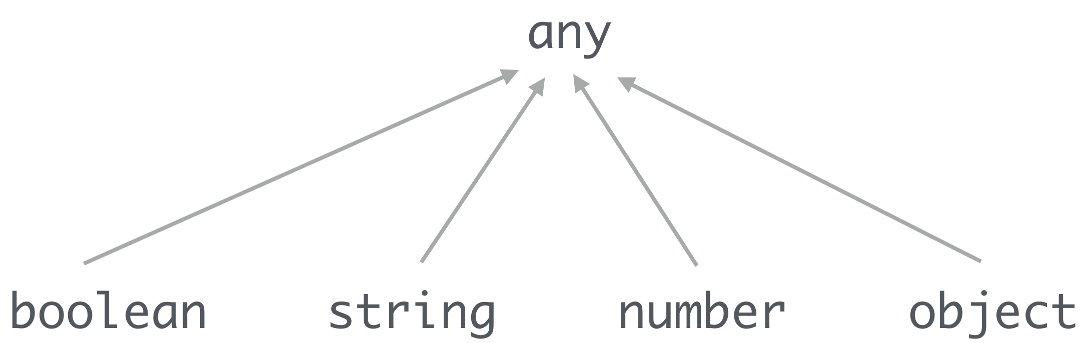
“I call it my billion-dollar mistake.
It was the invention of
the null reference in 1965.”

— C. A. R. Hoare

boolean string number object

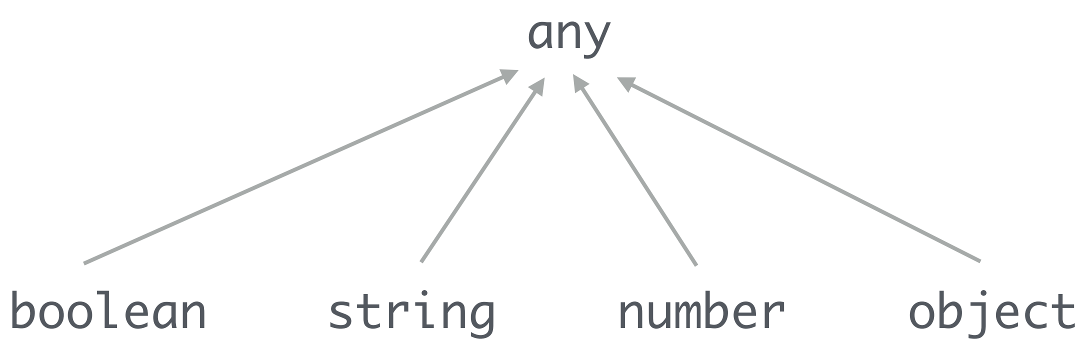
any

boolean string number object



any

boolean string number object

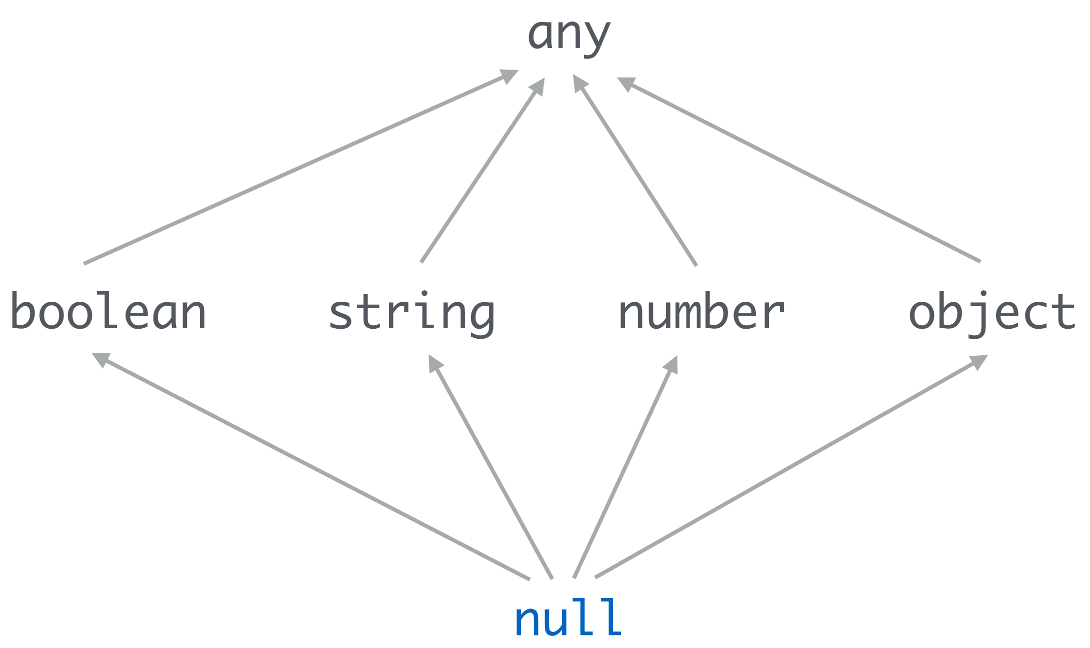


any

boolean string number object

null

TS



any

boolean

string

number

object

null

```
let names = ["Aseem", "Matt"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name.toUpperCase())
// > MATT
```

```
let names = ["Aseem", "Matt"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name.toUpperCase())
// > MATT
```

```
let names = ["Aseem"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name.toUpperCase())

// console.log(name.toUpperCase())
//                 ^
//
// TypeError: Cannot read property
// 'toUpperCase' of undefined
```

```
let names = ["Aseem", "Matt"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name.toUpperCase())
// > MATT
```

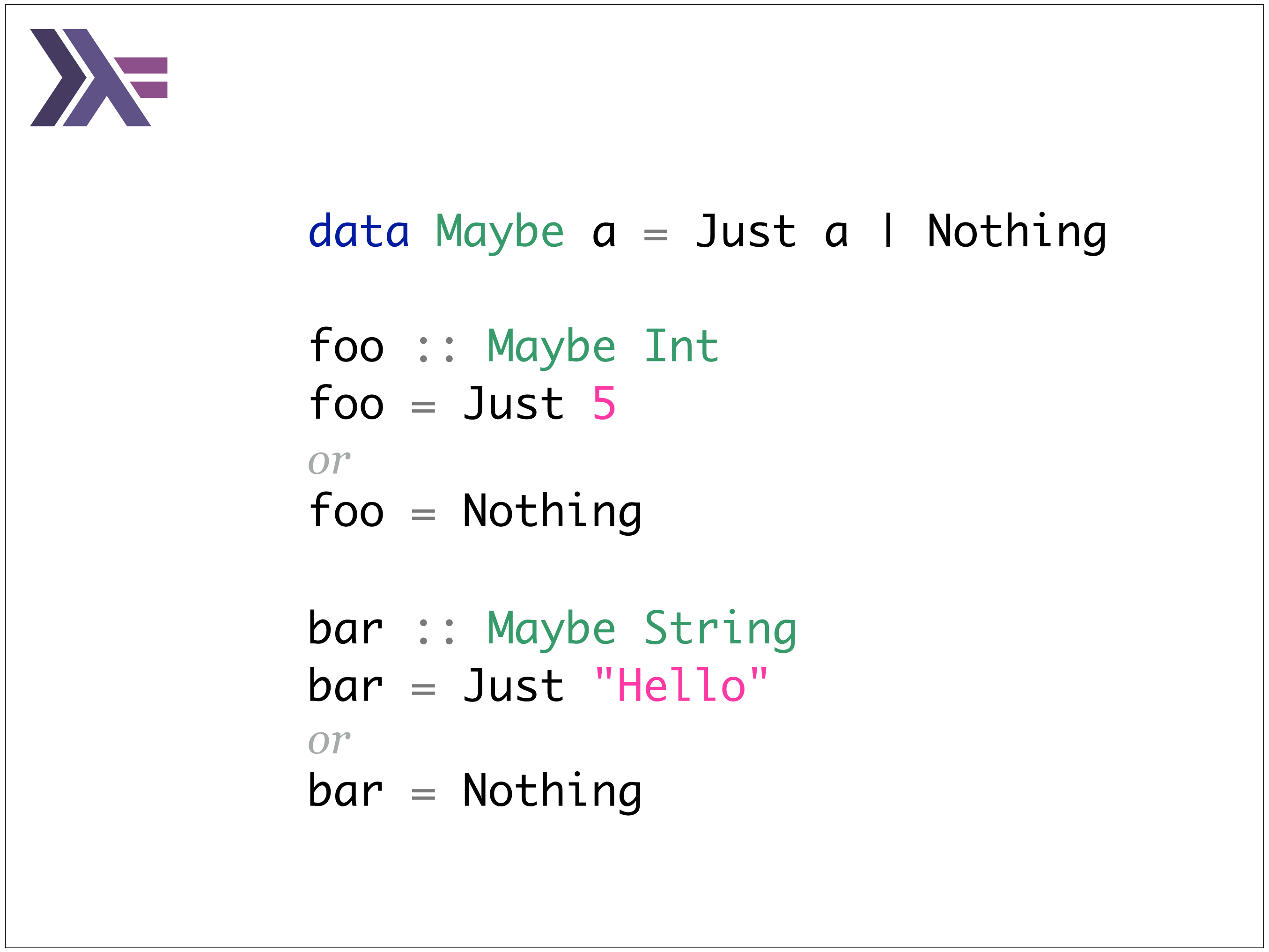
```
let names = ["Aseem"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name.toUpperCase())

// console.log(name.toUpperCase())
//                  ^
//
// TypeError: Cannot read property
// 'toUpperCase' of undefined
```

```
main = do
  let names = ["Aseem", "Matt"]
      isCool x = length x <= 4
      name = find isCool names
  print (toUpperCase name)

-- null.hs:5:22:
--  Coudn't match expected type ‘String’
--              with actual type ‘Maybe String’
--  In the first argument of ‘toUpperCase’,
--    namely ‘name’
--  In the first argument of ‘print’,
--    namely ‘(toUpperCase name)’
```

```
Array<A>.find(
  predicate: (value: A) => boolean
): A | null
```

```
find :: (a -> Bool) -> [a] -> Maybe a
```

```haskell
data Maybe a = Just a | Nothing
```



```haskell
data Maybe a = Just a | Nothing

foo :: Maybe Int
foo = Just 5
```

*or*

```haskell
foo = Nothing

bar :: Maybe String
bar = Just "Hello"
```

*or*

```haskell
bar = Nothing
```

```
let names = ["Aseem"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name ?
  name.toUpperCase() : "nuddin"
)
// nuddin
```

```javascript
let names = ["Aseem"]
let isCool = x => x.length <= 4
let name = names.find(isCool)
console.log(name ?
  name.toUpperCase() : "nuddin"
)
// nuddin
```

```haskell
main = do
  let names = ["Aseem"]
      isCool x = length x <= 4
      name = find isCool names
  print (case name of
      Just s  -> toUpperCase s
      Nothing -> "nuddin"
    )
-- nuddin
```

# Conclusion

Unhandled nulls can cause unexpected runtime errors.

Explicitly model the presence and absence of values and enforce handling of all cases.

# Types

# First-Class Compile-Time Type Safety

```haskell
data User = User
  { userId :: UserId
  , userEmail :: Email
  } deriving Show

newtype Email  = Email String deriving Show
newtype UserId = UserId String deriving Show

createUser :: UserId -> Email -> User
createUser userId userEmail = User { userId = userId, userEmail = userEmail }

-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)

{-
  types-user.hs:16:21:
    Couldn't match expected type ‘UserId’ with actual type ‘Email’
    In the first argument of ‘createUser’, namely ‘email’
    In the first argument of ‘print’, namely
      ‘(createUser email userId)’

  types-user.hs:16:27:
    Couldn't match expected type ‘Email’ with actual type ‘UserId’
    In the second argument of ‘createUser’, namely ‘userId’
    In the first argument of ‘print’, namely
      ‘(createUser email userId)’
-}
```

```
class User {
  private userId: string
  private userEmail: string

  constructor(userId: string, userEmail: string) {
    this.userId = userId
    this.userEmail = userEmail
  }
}

// Main
let email = 'daniel@fiftythree.com'
let userId = '3490'

console.log(new User(email, userId))
// User { userId: 'daniel@fiftythree.com', userEmail: '3490' }
```

```haskell
data User = User
  { userId :: UserId
  , userEmail :: Email
  } deriving Show
```

```typescript
class User {
  private userId: string
  private userEmail: string

  constructor(userId: string
              userEmail: string) {
    this.userId = userId
    this.userEmail = userEmail
  }
}
```

```typescript
// `deriving Show` is explicit generation
// of `Object.prototype.toString()`
```

```haskell
createUser :: UserId -> Email -> User
createUser userId userEmail = User
  { userId = userId
  , userEmail = userEmail
  }
```

```typescript
// function createUser(
//   userId: UserId,
//   userEmail: Email
// ): User
```

```typescript
class User {
  private userId: string
  private userEmail: string

  constructor(userId: string,
              userEmail: string) {
    this.userId = userId
    this.userEmail = userEmail
  }
}
```

```haskell
-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)

{-
  types-user.hs:16:21:
    Couldn't match expected type ‘UserId’
      with actual type ‘Email’
    In the first argument of ‘createUser’,
      namely ‘email’
    In the first argument of ‘print’,
      namely ‘(createUser email userId)’

  types-user.hs:16:27:
    Couldn't match expected type ‘Email’
      with actual type ‘UserId’
    In the second argument of ‘createUser’,
      namely ‘userId’
    In the first argument of ‘print’,
      namely ‘(createUser email userId)’
-}
```

```typescript
// Main
let email = 'daniel@fiftythree.com'
let userId = '3490'

console.log(new User(email, userId))
// User { userId: 'daniel@fiftythree.com',
//        userEmail: '3490' }
```

# (Awkward) ‘Solution’

```
class User {
  private userId: UserId
  private userEmail: Email

  constructor(userId: UserId, userEmail: Email) {
    this.userId = userId
    this.userEmail = userEmail
  }
}

type Email = string & {_emailBrand: any}
type UserId = string & {_userIdBrand: any}

// Main
let email = 'daniel@fiftythree.com' as Email
let userId = '3490' as UserId

console.log(new User(email, userId))
// Argument of type 'Email' is not assignable to parameter of type 'UserId'.
//   Type 'Email' is not assignable to type '{ _userIdBrand: any; }'.
//     Property '_userIdBrand' is missing in type 'Email'.
```

```haskell
newtype Email  = Email String deriving Show
newtype UserId = UserId String deriving Show

-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)
```

```haskell
newtype Email  = Email String deriving Show
newtype UserId = UserId String deriving Show

-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)
```

```typescript
type Email = string & {_emailBrand: any}
type UserId = string & {_userIdBrand: any}


// Main
let email = 'daniel@fiftythree.com' as Email
let userId = '3490' as UserId
console.log(new User(email, userId))
```

Language Feature

```haskell
newtype Email  = Email String deriving Show
newtype UserId = UserId String deriving Show

-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)
```

```typescript
type Email = string & {_emailBrand: any}
type UserId = string & {_userIdBrand: any}


// Main
let email = 'daniel@fiftythree.com' as Email
let userId = '3490' as UserId
console.log(new User(email, userId))
```

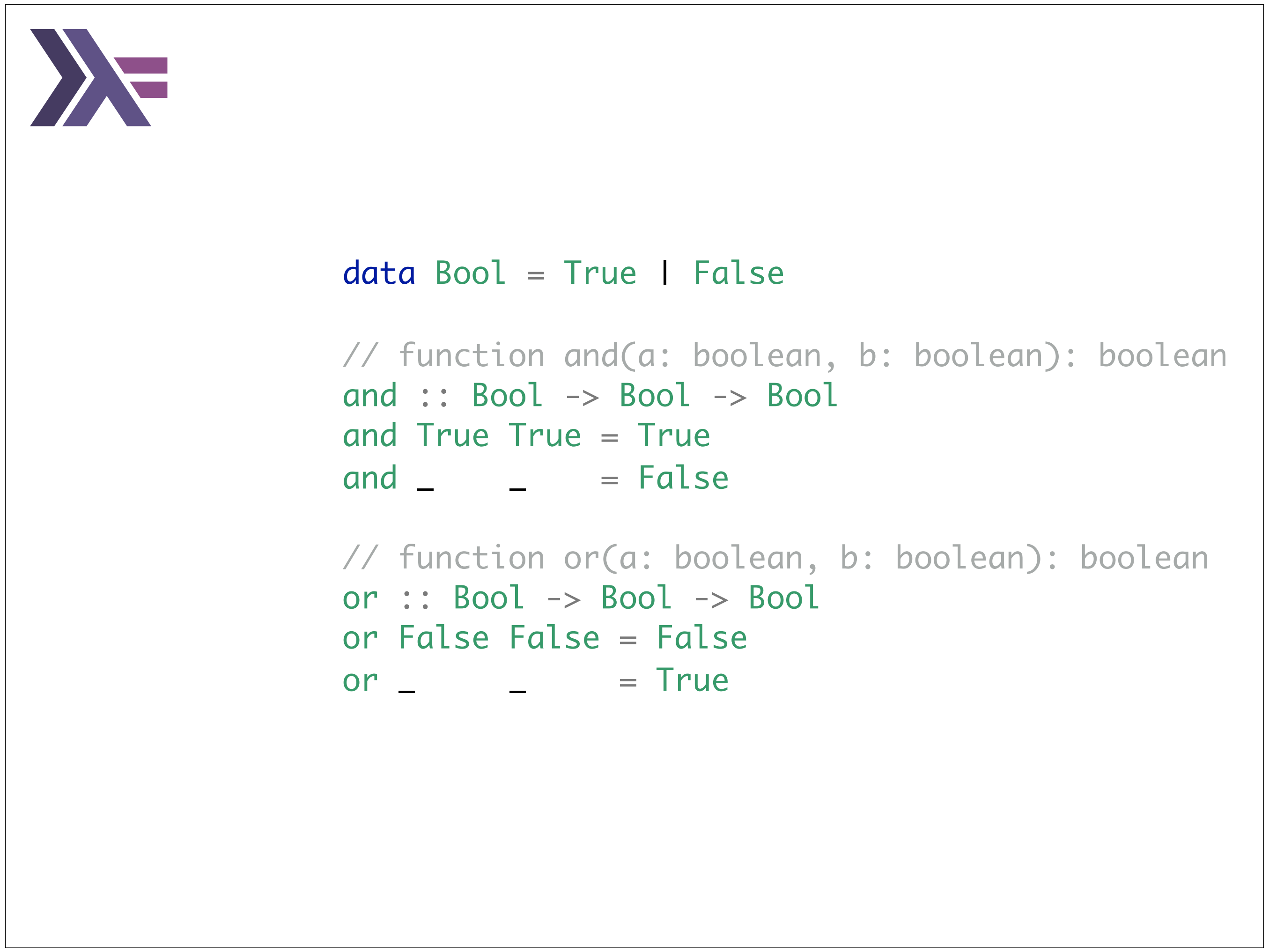
Language
Feature

```haskell
newtype Email  = Email String deriving Show
newtype UserId = UserId String deriving Show

-- Main
main = do
  let email = Email "daniel@fiftythree.com"
      userId = UserId "3490"
  print (createUser email userId)
```

Hack

```typescript
type Email = string & {_emailBrand: any}
type UserId = string & {_userIdBrand: any}


// Main
let email = 'daniel@fiftythree.com' as Email
let userId = '3490' as UserId
console.log(new User(email, userId))
```

# ‘Built-in’ Types



```
data Bool = True | False

// function and(a: boolean, b: boolean): boolean
and :: Bool -> Bool -> Bool
and True True = True
and _    _    = False

// function or(a: boolean, b: boolean): boolean
or :: Bool -> Bool -> Bool
or False False = False
or _     _     = True
```

This is a simplified illustration of to implement your own *Bool* type. The real Haskell definition is (only) slightly more involved.

```
data Bool = True | False
```

```
(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _    _    = False
```

```
// Define: (&&)
// Use: True && False
```

```
(||) :: Bool -> Bool -> Bool
(||) False False = False
(||) _     _     = True
```

# Security

“Make sure we *never*
store plaintext passwords
in our database.”

```haskell
newtype PlainTextPassword = PlainTextPassword String deriving Show
newtype HashedPassword = HashedPassword String deriving Show

getPassword :: IO PlainTextPassword
getPassword = do
  s <- getLine
  return (PlainTextPassword s)

hashPassword :: PlainTextPassword -> HashedPassword
hashPassword (PlainTextPassword s) = HashedPassword ((reverse s) ++ "$SALT$")

storePassword :: HashedPassword -> IO ()
storePassword (HashedPassword s) = putStrLn s

-- Main
main = do
  putStrLn "Enter password please:"
  p <- getPassword

  putStrLn "\nStored the following hashed password:"
  storePassword p

-- types-security.hs:21:17:
--     Couldn't match expected type ‘HashedPassword’
--                 with actual type ‘PlainTextPassword’
--     In the first argument of ‘storePassword’, namely ‘p’
--     In a stmt of a 'do' block: storePassword p
```

```haskell
newtype PlainTextPassword = PlainTextPassword String deriving Show
newtype HashedPassword = HashedPassword String deriving Show

getPassword :: IO PlainTextPassword
getPassword = do
  s <- getLine
  return (PlainTextPassword s)

hashPassword :: PlainTextPassword -> HashedPassword
hashPassword (PlainTextPassword s) = HashedPassword ((reverse s) ++ "$SALT$")

storePassword :: HashedPassword -> IO ()
storePassword (HashedPassword s) = putStrLn s

-- Main
main = do
  putStrLn "Enter password please:"
  p <- getPassword

  putStrLn "\nStored the following hashed password:"
  storePassword (hashPassword p) -- before: `storePassword p`

-- Enter password please:
-- passw0rd
--
-- Stored the following hashed password:
-- dr0wssap$SALT$
```

# Conclusion

Types can help prevent many errors at compile-time.
They are a versatile and powerful tool to model your domain.

# Abstraction
# &
# Type Classes

map

```
console.log([1, 2, 3].map(x => x * 3))
// [3, 6, 9]
// Array<A>.map<B>(fn: (value: A) => B): Array<B>
console.log([1, 2, 3].map(x => x * 3))
// [3, 6, 9]
```

```haskell
main = do
  -- map :: (a -> b) -> [a] -> [b]
  print (map (\x -> x * 3) [1, 2, 3])
  -- [3, 6, 9]
```

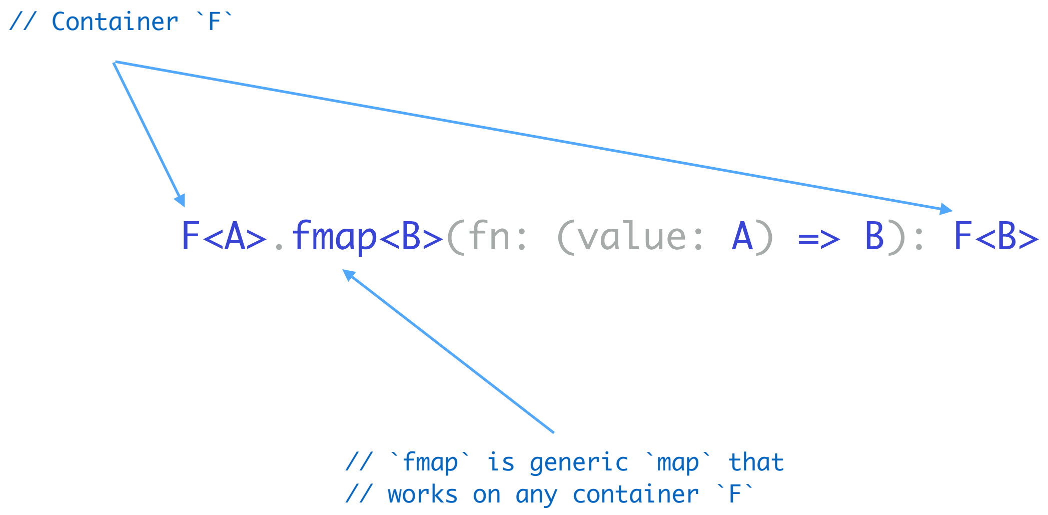
```
main = do
  -- map :: (a -> b) -> [a] -> [b]
  print (map (\x -> x * 3) [1, 2, 3])
  -- [3, 6, 9]
```

```
main = do
  -- map :: (a -> b) -> [a] -> [b]
  print (map (3*) [1, 2, 3])
  -- [3, 6, 9]
```

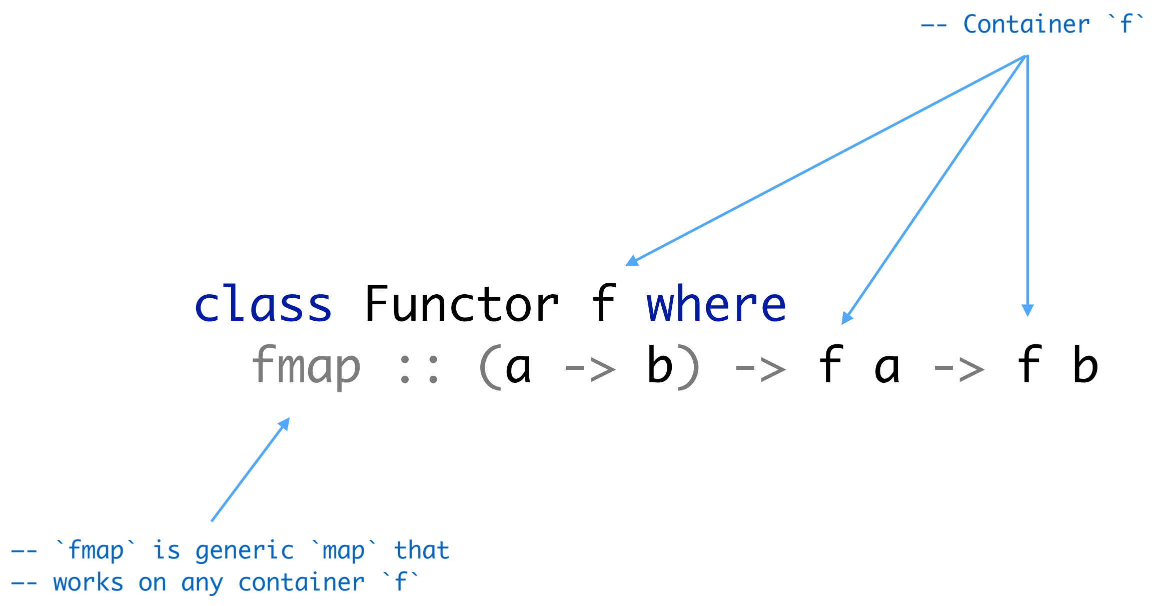
```haskell
main = do
  -- map :: (a -> b) -> [a] -> [b]
  print (map (3*) [1, 2, 3])
  -- [3, 6, 9]
```

```typescript
// Array<A>.map<B>(
//   fn: (value: A) => B
// ): Array<B>
console.log([1, 2, 3].map(x => x * 3))
// [3, 6, 9]
```

```
Array<A>.map<B>(fn: (value: A) => B): Array<B>
```



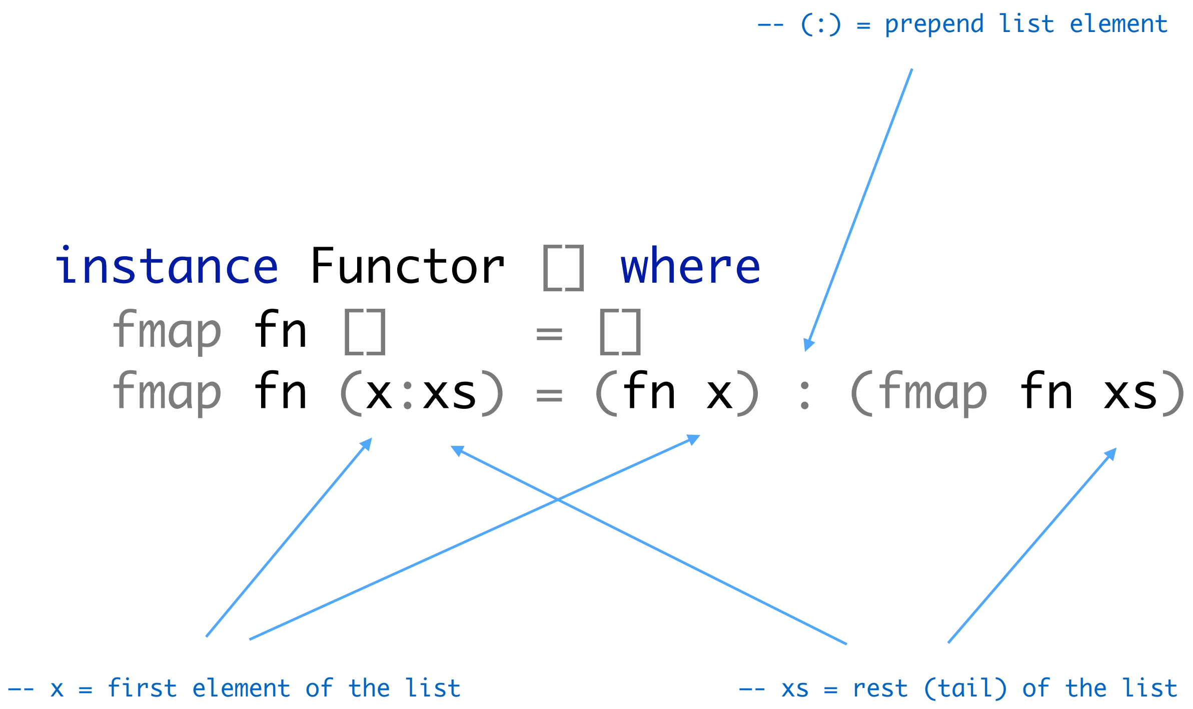
```
// Container `F`

F<A>.fmap<B>(fn: (value: A) => B): F<B>

// `fmap` is generic `map` that
// works on any container `F`
```



```
-- Container `f`
```

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
-- `fmap` is generic `map` that
-- works on any container `f`
```



-- (:) = prepend list element

```haskell
instance Functor [] where
  fmap fn []     = []
  fmap fn (x:xs) = (fn x) : (fmap fn xs)
```

-- x = first element of the list

-- xs = rest (tail) of the list

```
instance Functor [] where
  fmap = map
```

```
instance Functor Maybe where
  fmap fn Nothing  = Nothing
  fmap fn (Just x) = Just (fn x)
```

```haskell
main = do
  -- List
  print (fmap (3*) [1, 2, 3])
  -- > [3, 6, 9]

  -- Maybe
  print (fmap (3*) Nothing)
  -- > Nothing
  print (fmap (3*) (Just 2))
  -- > Just 6

  -- IO
  -- getLine :: IO String
  putStrLn "\nWhat is your name?"
  message <- fmap ("Hello, " ++) getLine
  putStrLn message
  -- > What is your name?
  -- > Daniel
  -- > "Hello, Daniel"

  -- Async
  putStrLn "\nSay something…"
  asyncPrompt <- async getLine
  asyncMessage <- wait (fmap ("Async: " ++) asyncPrompt)
  putStrLn asyncMessage
  -- > Say something…
  -- > Yo yo
  -- > Async: Yo yo
```

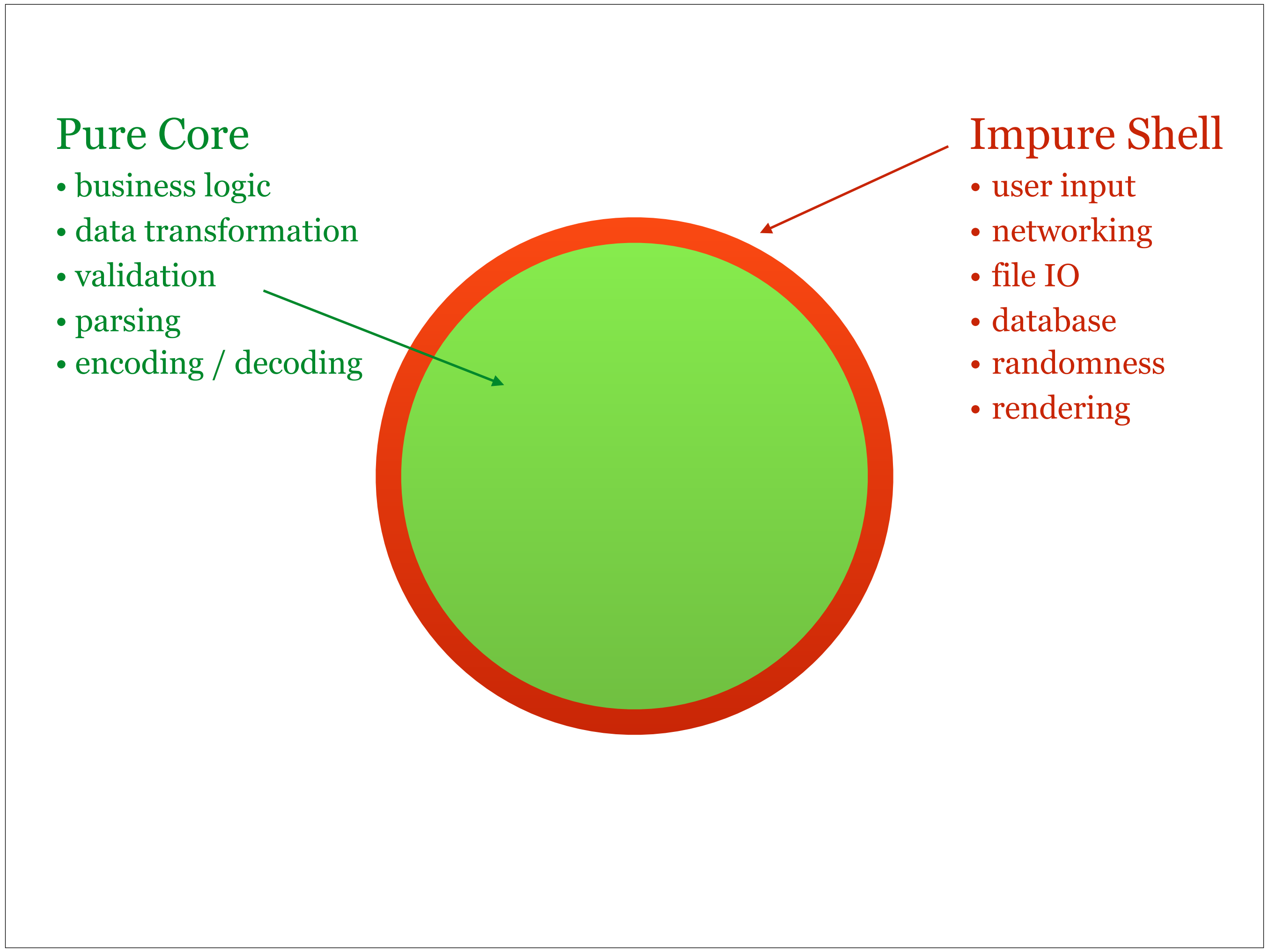
# Conclusion

Expressive languages allow developers to describe better abstractions.

Type classes are a mechanism for abstracting common behaviors between different types.

# Food for Thought

# Impure Shell
# &
# Pure Core



Pure Core

- business logic
- data transformation
- validation
- parsing
- encoding / decoding

Impure Shell

- user input
- networking
- file IO
- database
- randomness
- rendering

Pure Core

- pure computations (no external input besides arguments and no side-effects)
- immutable data
- testable because a pure function returns the same output for a set of specific inputs

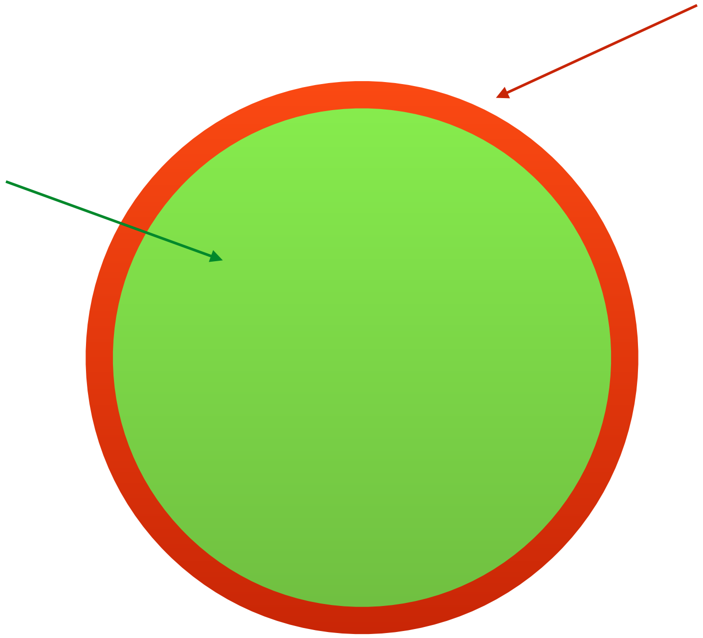
Impure Shell

- Side-effects
- Mutation
- Hard to test

# Example: Compiler

## Pure Core

- lexical analysis
- syntax analysis
- type checking
- optimize code
- generate code



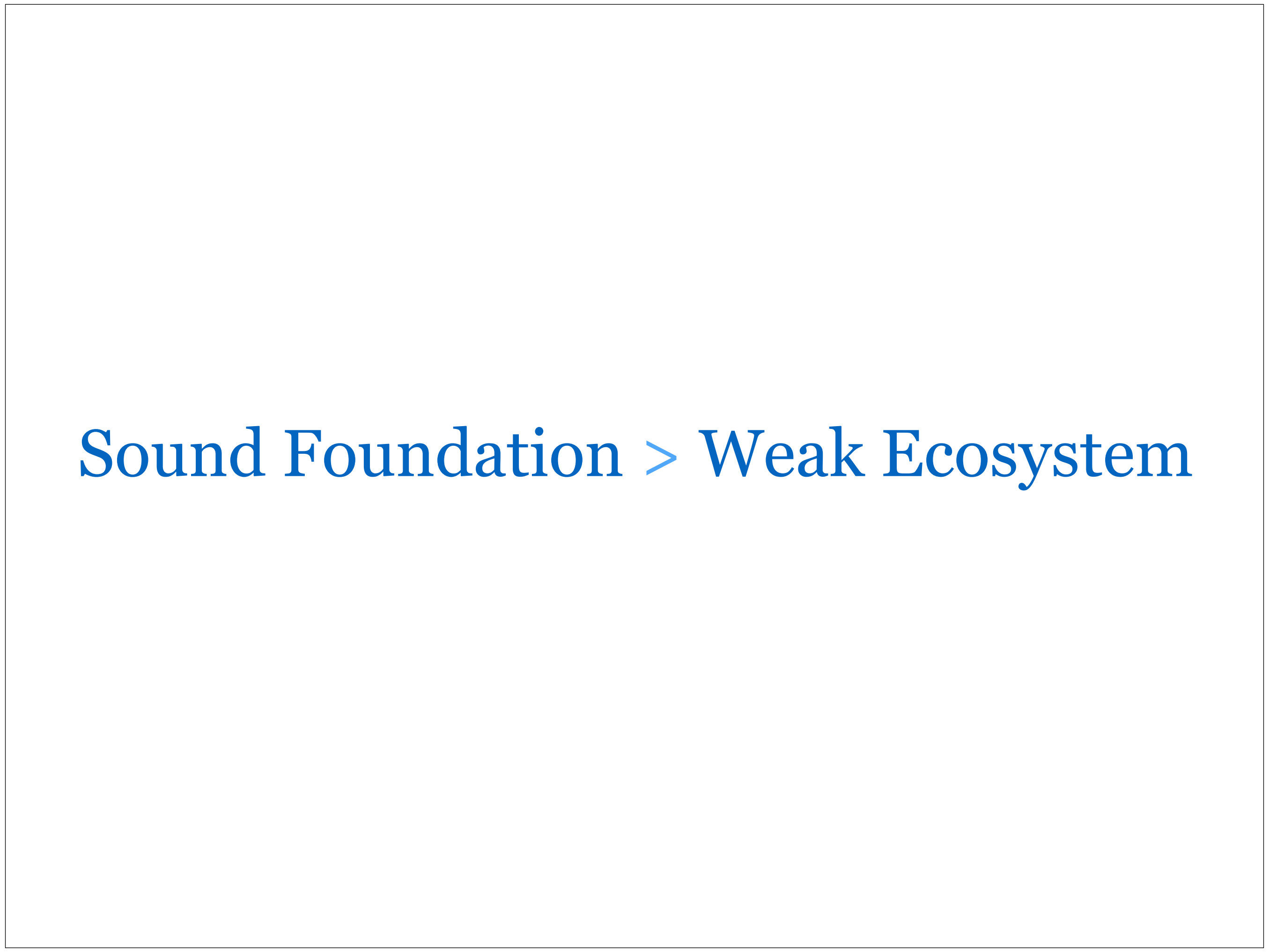
## Impure Shell

- read CLI options
- read environment variables
- read source files
- write binary

PARENTAL

**ADVISORY**

CONTROVERSIAL CONTENT

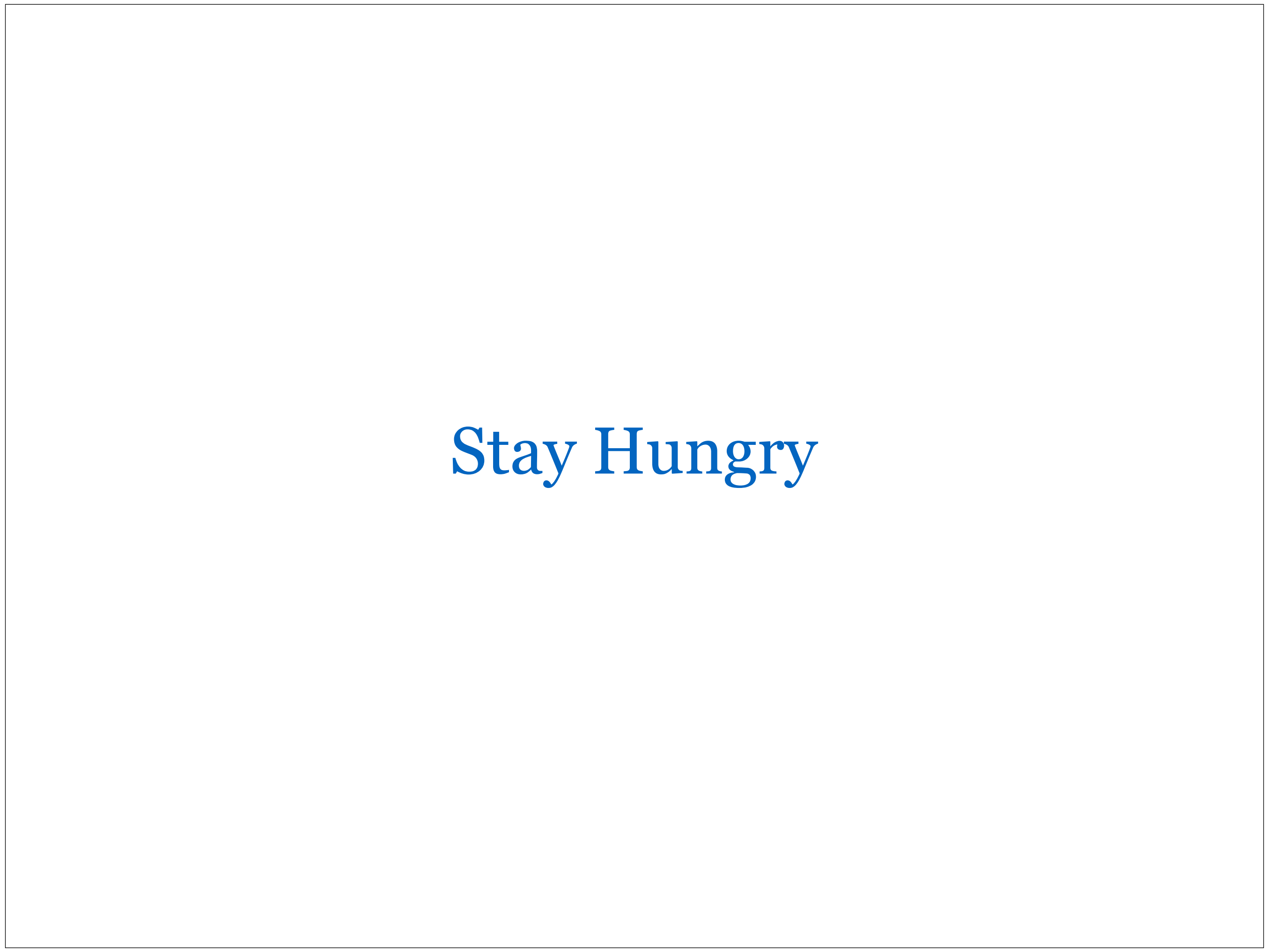


# Sound Foundation > Weak Ecosystem

Compared to many imperative languages, functional languages have a sound foundation with weaknesses in their ecosystem, e.g. tooling, documentation, education, etc.

However, no matter how good your tooling/libraries, etc. are, if mutation, `null`, side-effects, etc. are at the core of your foundation, you will always struggle (runtime errors, difficulty with parallelism/concurrency/multicore, lack of STM, etc.).

On the other hand, tooling for a sound system can be improved through benevolent volunteers, industry adoption, etc.



# Stay Hungry

If you ever decide to learn a new language, instead of picking another imperative language such as Go, even Swift, etc., which are very similar to what you have probably been using all your life except for a few new concepts, pick something with a vastly different approach, e.g. a functional programming language such as Haskell, Standard ML, OCaml, PureScript, Elm. If FP is not your thing, at least pick something like Prolog (logic programming), Matlab (array language), etc.

“The only thing necessary for the triumph of [bad technology] is for good men to do nothing.”

# Q&A

“All [bad technology] needs to gain a foothold is for people of good conscience to remain silent.”

Thank you. Does anyone have any questions?