propaganda version

Fix your most common mistake with this one simple trick

academic version

Exceptions, NullObjects, promises and the Maybe monad

“

> Show me how you handle errors
> and I'll tell you what programmer you are.

schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/10_Error_Handling

Chapter 1

Java

Do you know all the possible exceptions in your app?

In Java, you *kinda* do.
You’re *kinda* forced to.

```
public foo() throws SomethingTerrible {
   …
}

public bar() {
  someObject.foo()
}
```

⟹ Unhandled exception type SomethingTerrible

```
public foo() throws SomethingTerrible {
  …
}

public bar() throws SomethingTerrible {
  someObject.foo()
}
```

```
public foo() throws SomethingTerrible {
  …
}

public bar() {
  try {
    someObject.foo()
  } catch (SomethingTerrible error) {
    …
  }
}
```

problems

You have to explicitly pass all checked exceptions.
It’s not very type aware.

People can still use “unchecked” exceptions
(RuntimeException, Error and subclasses) and they do

Makes people assume all problems
should be modelled as exceptions.

Chapter 2

Ruby

In Ruby, you don’t.
Bang is *supposed* to indicate *something*.

But again, what’s the difference between something expected, and exception you treat and an exception you catch in a catch all?

What is the most common problem in our apps?

in-app

```
TypeError: Cannot read property 'triggerCallback' of undefined
NoMethodError: undefined method `[]' for nil:NilClass
NoMethodError: undefined method `include?' for nil:NilClass
```

myKlarna

```
NoMethodError: undefined method `[]' for nil:NilClass
NoMethodError: undefined method `downcase' for nil:NilClass
NoMethodError: undefined method `join' for nil:NilClass
NoMethodError: undefined method `card_data' for nil:NilClass
TypeError: Cannot read property '_currentElement' of null
TypeError: Cannot read property 'paymentMethod' of null
TypeError: Cannot read property 'id' of null
undefined: Unable to get property 'paymentMethod' of undefined or null reference
```

SLOT

```
NoMethodError: undefined method `to_hash' for nil:NilClass
NoMethodError: undefined method `denied?' for nil:NilClass
NoMethodError: undefined method `[]' for nil:NilClass
```

DG

```
TypeError: Cannot read property 'inserted' of undefined
TypeError: Cannot read property 'phoneVerificationToken' of null
TypeError: Cannot read property 'match' of null
```

KCO

```
TypeError: Cannot read property 'name' of undefined
TypeError: Cannot read property 'value' of null
TypeError: Cannot read property 'PURCHASE_COUNTRY' of undefined
TypeError: Cannot read property 'postMessage' of null
TypeError: Cannot call method 'getItem' of null
```

Something that may not be there.

```
user.andand.name
user.try(:name)
```

solution

NullObject & duck typing

```
class NullUser
  def name; "Anonymous" end
end

def current_user
  User.find(session[:user_id]) || NullUser.new
end

<h1>
 Hello <%= user.name %>!
</h1>
```

problem

since in a regular app
most entities may not be there
we end up with multiplication of entities:
NullEverything

```
<h1>
 <% if user %>
   Hello <%= user.name %>!
 <% else %>
   <a href="/login">Log in</a>
 <% end %>
</h1>
```

```
class NullObject
  def nil?; true; end
  def present?; false; end
  def empty?; true; end
  def !; true; end
  def method_missing(*args, &block)
    self
  end
end

class NullUser < NullObject; end
```

```
<h1>
 <% if user %>
   Hello <%= user.name %>!
 <% else %>
   <a href="/login">Log in</a>
 <% end %>
</h1>

==> "Hello #<NullUser:0x007fd4ca1dd188>"
```

```
if !!user

if user.nil?
```

“

> If we’re trying to coerce a homemade object into acting falsey, we may be chasing a vain ideal. …it is *almost* always possible to transform code from typecasing conditionals to duck-typed polymorphic method calls.

devblog.avdi.org/2011/05/30/null-objects-and-falsiness

```ruby
require 'delegate'

class UserPresenter < SimpleDelegator
  def login_status; "You are logged in as #{name}" end
end

class NullUserPresenter < SimpleDelegator
  def login_status; "You are not logged in" end
end

def Present(object)
  Object.const_get("#{object.class.name}Presenter").new(object)
end

Present(current_user).login_status
```

MOTHER OF GOD!!!

final problem

People end up not doing it.

The language doesn’t force them.

It’s faster to finish your ticket by adding one *andand* here and there.

Chapter 3

Haskell

greenbelt

Chapter 3.1

Haskell’s fundamentals

Haskell is a functional language

the two definitions

A language
where functions are first class citizens

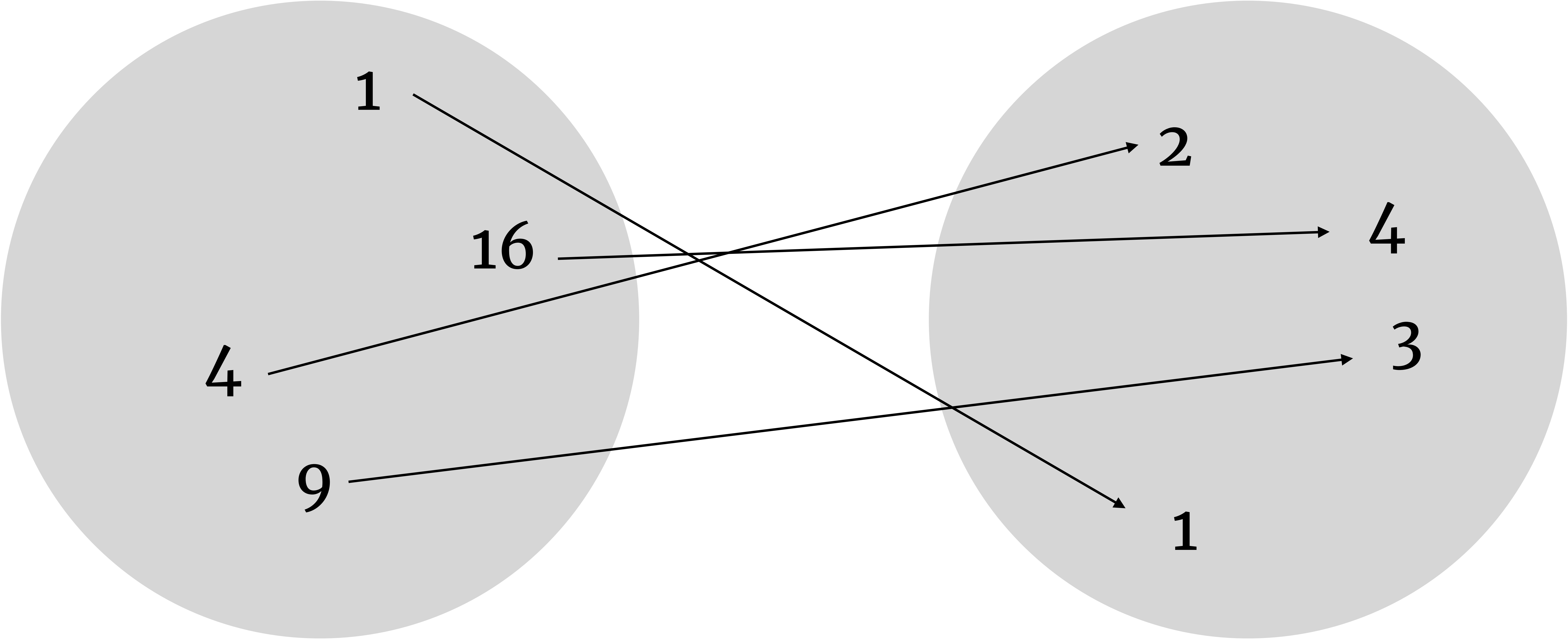
A language
where you code *only* with *mathematical* functions

function in imperative programming

subroutine (another fancy goto)

function in *mathematical* sense

maps a value from one domain to another


1
16
4
9
2
4
3
1

In Haskell, everything is pure.

* there's System.IO.Unsafe and the infamous unsafePerformIO

```
sum 3 4
⟹ 7
sum 3 4
⟹ 7
sum 3 4
⟹ 7
```

```
[-∞, -∞] ⟹ -∞
…
[-1, -1] ⟹ -2
[-1,  0] ⟹ -1
[ 0, -1] ⟹ -1
[ 0,  0] ⟹ -1
[ 0,  1] ⟹  1
[ 1,  0] ⟹  1
[ 1,  1] ⟹  2
…
[ ∞,  ∞] ⟹  ∞
```

But wait… how can *everything* be pure?

Chapter 3.2

Determinism

```
rand
⟹ 0.23781878247847277
rand
⟹ 0.07496989423824807
rand
⟹ 0.9155843604468722
```

pause for a computer science question

Is random a pure function?

“Real” randoms

Depend on physical phenomena.
Like nuclear decay.

“Pseudo” randoms

Generally depend on a seed.
Like the current time.

```
Array.new(10 ** 6){
  Random.new(10).rand(10)
}.uniq
⟹ [9]

Array.new(10 ** 6){
  Random.new(293857).rand(10)
}.uniq
⟹ [2]
```

```
random :: PhysicalPhenomena → Float
```

pause for a philosophical question

Is there such a thing as a random at all?

```
random :: SnapshotOfTheUniverse → Float
```

Chapter 3.3

Haskell’s IO

```
random :: Float
random :: RealWorld → (RealWorld, Float)
random :: IO Float
randomIO :: Random a ⇒ IO a

getLine :: String
getLine :: RealWorld → (RealWorld, String)
getLine :: IO String

putStrLn :: String → ()
putStrLn :: String → RealWorld → (RealWorld, ())
putStrLn :: String → IO ()
```

```
main =
  operation1 >>= \result1 ->
    operation2 result1 >>= \result2 ->
      operation3 result2 >>= \result3 ->
        operation4 result3 >>= \result4 ->
          …
                   …
                            operation23849823 result23849822
```

```
main = do
  result1 ← operation1
  result2 ← operation2 result1
  result3 ← operation3 result2
  result4 ← operation4 result3
  …
  …
  operation23849823 (result23849822)
```

```
main do
 putStrLn "What is your name?"
 name ← getLine
 putStrLn "Hello " ++ name
```

In Haskell there are no exceptions
built in the language.

```
handle :: Exception e ⇒ (e → IO a) → IO a → IO a
catch  :: Exception e ⇒ IO a → (e → IO a) → IO a
try    :: Exception e ⇒ IO a → IO (Either e a)
```

```
returnEmpty :: SomeException → IO String
returnEmpty _ = return ""

main do
 line ← catch getLine returnEmpty
 putStrLn line
```

Chapter 3.4

Maybe

IO	A value of type IO a is a computation which, when performed, does some I/O before returning a value of type a. There is really only one way to "perform" an I/O action: bind it to Main.main in your program. When your program is run, the I/O will be performed. It isn't possible to perform I/O from an arbitrary function, unless that function is itself in the IO monad and called at some point, directly or indirectly, from Main.main.
Maybe	The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing). Using Maybe is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.
Either	The Either type represents values with two possibilities: a value of type Either a b is either Left a or Right b.
AccValidation	The AccValidation data type is isomorphic to Either, but has an instance of Applicative that accumulates on the error side. That is to say, if two (or more) errors are encountered, they are appended using a Semigroup operation.
Cont	The Continuation monad represents computations in continuation-passing style (CPS). In continuation-passing style function result is not returned, but instead is passed to another function, received as a parameter (continuation).
Except	Computations which may fail or throw exceptions. Failure records information about the cause/location of the failure. Failure values bypass the bound function, other values are used as inputs to the bound function.

functional solution

algebraic data types

```
data Bool = True | False
data Int = -2147483648 | … | -1 | 0 | 1 | … | 2147483647
data Maybe a = Just a | Nothing
data User = User { name :: String }

login :: User -> String
login user = "Hello " ++ name user

login (User "Joe")
==> "Hello Joe"
```

```haskell
data User = User { name :: String }

login :: Maybe User -> String
login (Just user) = "Hello " ++ name user
login Nothing = "You have to login"

login (Just (User "Joe"))
==> "Hello Joe"

login Nothing
==> "You have to login"
```

```haskell
import Data.Maybe

data User = User { name :: String }

currentUser :: Maybe User -> User
currentUser = fromMaybe (User "Anonymous")

great :: User -> String
great user = "Hello " ++ name user
```

```haskell
import Data.Maybe

data User = User { name :: String }


currentUser = fromMaybe (User "Anonymous")


great user = "Hello " ++ name user
```

```haskell
divide :: Float → Float → Maybe Float
divide x 0 = Nothing
divide x y = Just (x / y)

calc :: Maybe Float
calc = do
  a ← divide 10 2
  b ← divide a 1
  c ← divide b 5
  return c ⟹ Just 1.0
```

```
calc :: Maybe Float
calc = do
  a ← divide 10 2
  b ← divide a 0
  c ← divide b 5
  return c ⟹ Nothing
```

Chapter 4

Back to Ruby

andand’s README:

“The Maybe Monad in idiomatic Ruby”.

“A few people have pointed out that Object#andand
is similar to Haskell’s Maybe monad.”

Lies.

```
nil.andand.foo.bar.baz
⟹ NoMethodError: undefined method 'bar' for nil:NilClass

nil.andand.foo.andand.bar.andand.baz
⟹ nil

nil.andand.length > 1
⟹ NoMethodError: undefined method '>' for nil:NilClass
```

```
user = User.new(username: "Joe")
Maybe(user).username.downcase
⟹ #<Monadic::Just:0x… @value="joe">
Maybe(user).username.downcase.fetch
⟹ "joe"
```

```
user = nil
Maybe(user).username.downcase
⟹ Monadic::Nothing
Maybe(user).username.downcase.fetch
⟹ Monadic::Nothing
Maybe(nil).downcase.or("anonymous")
⟹ #<Monadic::Just:0x… @value="anonymous">
Maybe(nil).downcase.or("anonymous").fetch
"anonymous"
```

```
def current_user
  user = User.fetch_from_session(session[:user_id])
  Maybe(user)
end
```

```
nil && "Oooops"
⟹ nil

Maybe(nil) && "Oooops"
⟹ "Oooops"

Maybe(nil).fetch && "Oooops"
⟹ "Oooops"

Maybe(nil).or(nil).fetch && "Oooops"
⟹ "Oooops"
```

“

If we’re trying to coerce a homemade object into acting falsey, we may be chasing a vain ideal.

devblog.avdi.org/2011/05/30/null-objects-and-falsiness

```
def current_user
  user = User.fetch_from_session(session[:user_id])
  Maybe(user).or(User::Anonymous.new)
end
```

Chapter 5

JS

Now, in JS, what is a chainable thing
that passes values or exceptions along?

```
const succ = (value) => value + 1

Promise
  .resolve(5)
  .then(succ)
  .then(succ)
  .then(console.log) ==> 7
  .catch(e => console.log("ERROR", e))
```

```
const succ = (value) => value + 1

Promise
  .resolve(null)
  .then(succ)
  .then(succ)
  .then(console.log)
  .catch(e => console.log("ERROR", e))
```

```
const succ = (value) => value + 1

Promise
  .resolve(null)
  .then(succ)
  .then(succ)
  .then(console.log) ==> 2 WTF ???
  .catch(e => console.log("ERROR", e))
```

```
const succ = (value) => value + 1

Promise
  .resolve(Just(5))
  .then(ap(succ))
  .then(ap(succ))
  .then(console.log) ==> { isValue: true, val: 7 }
  .catch(e => console.log("ERROR", e))
```

```
const succ = (value) => value + 1

Promise
  .resolve(Nothing())
  .then(ap(succ))
  .then(ap(succ))
  .then(console.log) ==> { isValue: false, val: null }
  .catch(e => console.log("ERROR", e))
```

```
const length = (value) ⇒ value.length

Promise
  .resolve("Hello")
  .then(length)
  .then(console.log) ⟹ 4
  .catch(e ⇒ console.log("ERROR", e))
```

```
const length = (value) => value.length

Promise
  .resolve(null)
  .then(length)
  .then(console.log) <== this doesn’t get executed
  .catch(e => console.log("ERROR", e))
==> ERROR TypeError: Cannot read property 'length' of null
```

```
const length = (value) => value.length

Promise
  .resolve(Nothing())
  .then(ap(length))
  .then(console.log) <== { isValue: false, val: null }
  .catch(e => console.log("ERROR", e))
```

```
const length = (value) => value.length
const recover = (v) => v.isNothing() ? Just(42) : v

Promise
  .resolve(Nothing())
  .then(ap(length))
  .then(recover)
  .then(console.log) <== { isValue: true, val: 42}
  .catch(e => console.log("ERROR", e))
```

```
Do(function* () {
  const a = yield Just(7)
  const b = yield Just(a + 9)
  return b
}, Maybe).val ⟹ 16

Do(function* () {
  const a = yield Just(7)
  const q = yield Nothing()
  const b = yield Just(a + 9)
  return b
}, Maybe).val ⟹ null
```

take aways

Every problem deserve it’s own understanding.

TypeError and NoMethodError are too common.
They mean something *expected* is not there.
Therefore they should not be treated as exceptions.

If you’re doing Ruby, resist try/andand.
Start writing more NullObjects. Check Monadic.

Most languagues, but particularly JS,
benefits from functional concepts.

Haskell is an everlasting source of inspiration.

resources

Ruby

devblog.avdi.org/2011/05/30/null-objects-and-falsiness (NullObject pattern in Ruby)

github.com/pzol/monadic (Some monads for Ruby)

github.com/rap1ds/ruby-possibly (Maybe only)

JS

el-tramo.be/blog/async-monad (Explains how async solves callback hell)

github.com/fantasyland/fantasy-land (Algebraic types for ES6)

github.com/russellmcc/fantasydo (Do notation for fantasy-land)

ecma-international.org/ecma-262/6.0 (Enumerate specs)

folktalejs.org, cwmyers.github.io/monet.js (Some monads for JS, fantasy-land compliant)

Haskell

youtube.com/watch?v=z0N1aZ6SnBk (Erik Meijer explains Haskell's IO, around 22')

dev.stephendiehl.com/hask (Best intro to modern day Haskell)

schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/10_Error_Handling (Error handling)