Functional Programming: *The Enterprise Edition*

...

Narek Asadorian

ScalaIO | 30 Oct 2018

Slides+Code: https://git.io/vh4VB

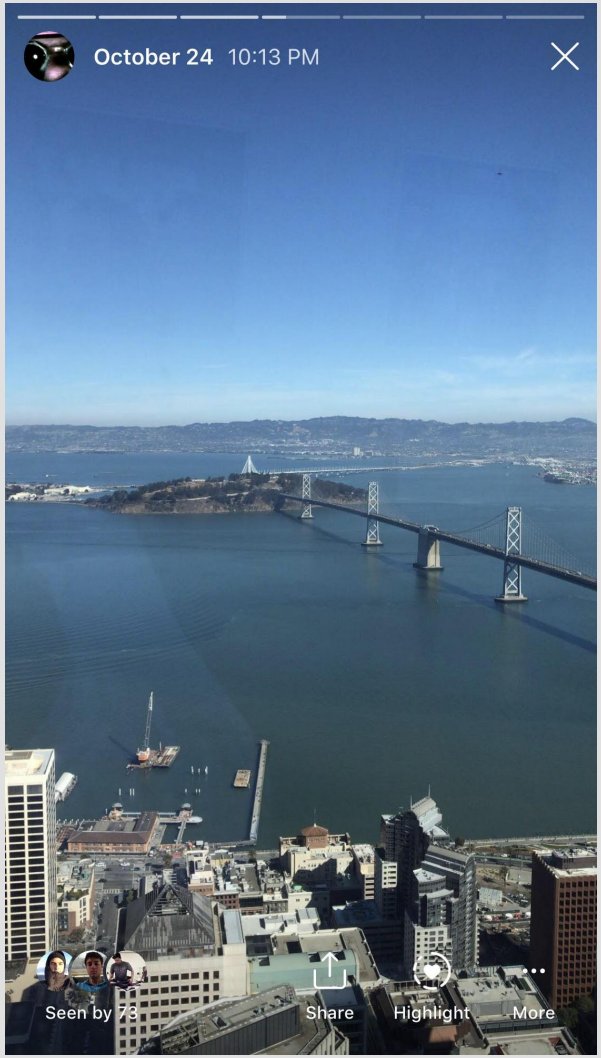
I’m Narek.

Senior Software Engineer

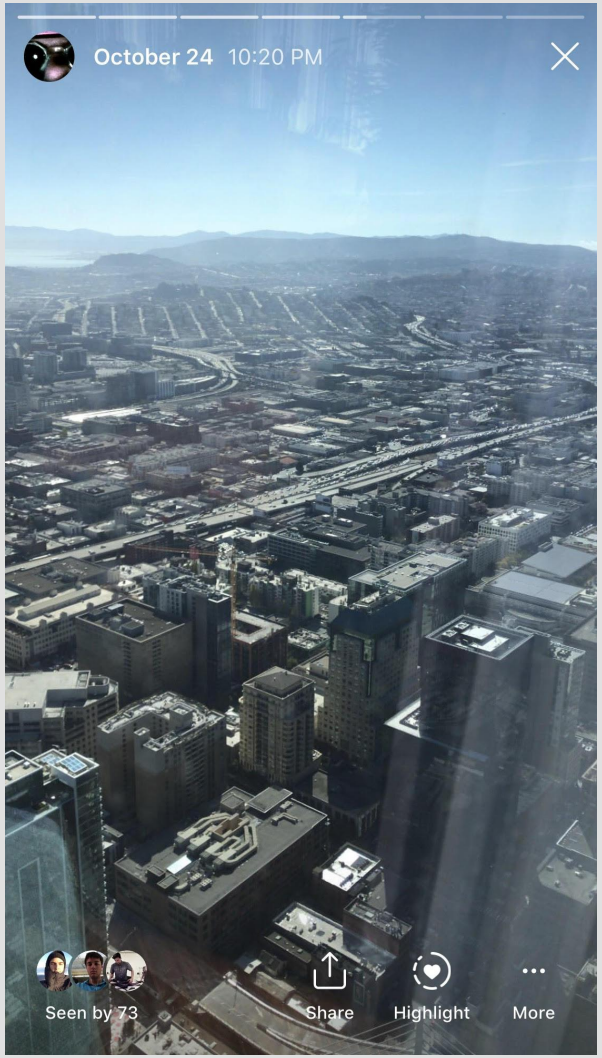
@portal_narlish

Scala & FP advocate

=> Streaming pipelines, microservices & data platform for email apps
=> Millions of customer events per day
=> **We're hiring...**


October 24 10:13 PM
Seen by 73
Share
Highlight
More


October 24 10:12 PM
Seen by 79
Share
Highlight
More


October 24 10:20 PM
Seen by 73
Share
Highlight
More

Roll call!

Are you writing Scala at work?

Is it purely functional?

Enterprise Functional Programming

Oxymoron?

def:

Enterprise Programming

“...display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes.”

- Martin Fowler

def:

Enterprise Programming

Optimizing for:

1. **Correctness**
2. **Reliability**
3. **Changing business needs**

def: Functional Programming

“programming with functions”

“no side effects”

“no mutability, no shared state”

def:

Functional Programming

Optimizing for:

1. **Composability**
2. **Purity**
3. **Generic abstractions**

What are the generalizations associated with the two?

Enterprise Code

- Impure
- Complicated logic
- Integration tests
- OOP hell
- 10-100 normal engineers

Functional Code

- Pure
- Elegant, clean
- Lawful and provable
- Perfect abstraction
- A few geniuses in the ivory tower

Enterprise Languages

- Java
- Javascript
- PHP
- Python
- C#, C++
- Scala

Functional Languages

- Haskell
- OCaml
- Clojure
- Erlang
- F#
- Scala

Thesis:

We'll never be perfect, but pure functional enterprise code **is absolutely possible**.

Functional abstractions are well suited for commercial software.

Scala forms a bridge between these worlds.

Caveats:

Functional programming as of yet remains a niche compared to the imperative style.

Teams using Scala are not necessarily invested in FP e.g. “Java in Scala”

“Naysayers” to FP are ubiquitous...

Bringing up FP at work shouldn't have to feel like this...

image: Twitter

But that's often the case because the imperative mindset is the default...

Why tho?
(ಠ_ಠ)

Why? Blame COBOL.

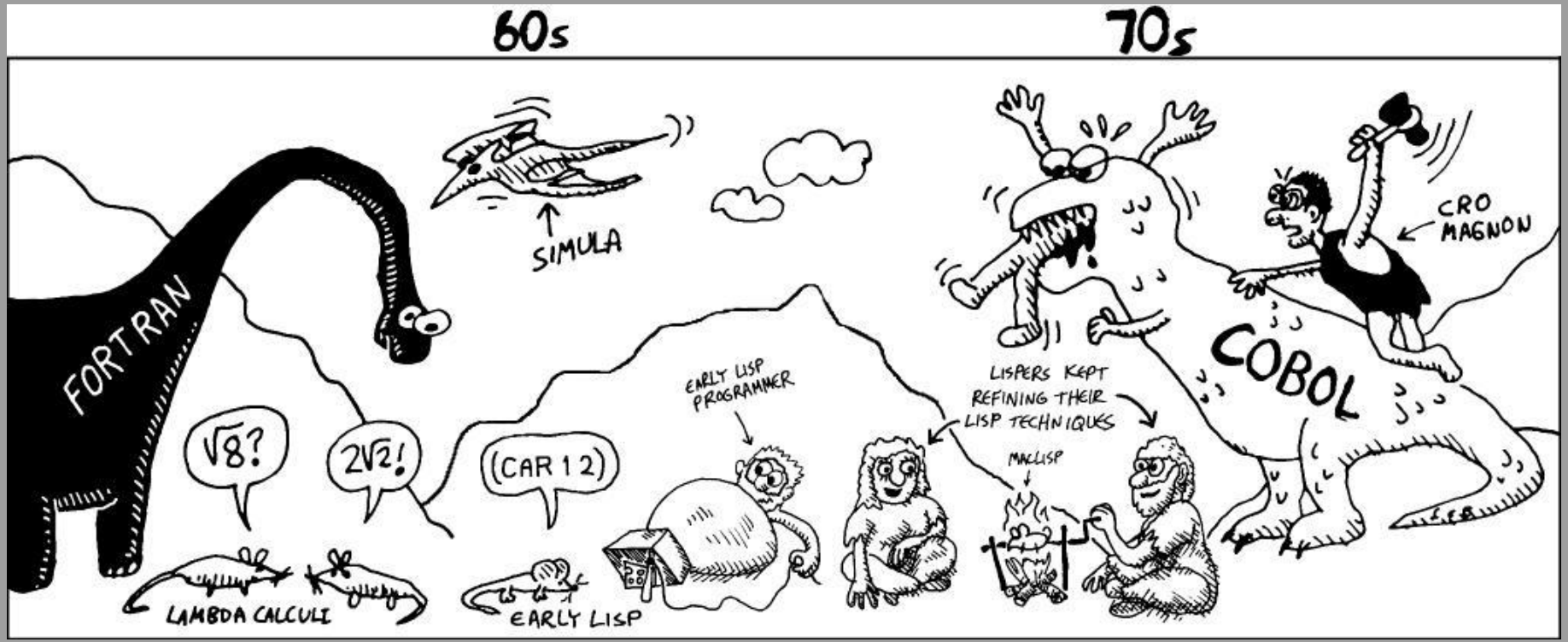


image: “Land of LISP” by Conrad Barski

Why? Blame universities.

- **C**, **Java** or **Python** taught first
- Functional concepts only play a minor role in CS education
- FP langs are “too high level” for teaching fundamentals of CE

Why? Blame fear.

- Programmers are afraid
  - “Scary math stuff”
  - Imposter syndrome
- Managers are afraid
  - *How will we maintain this alien code when you leave?*
- Some people are also outright *against* FP and fight it...

We want to show that functional programming is useful without scaring people...

The rest of this talk...

1. Applied functional abstractions for use in enterprise programming

2. Strategies for introducing and fostering *functional thinking* in commercial software development

More Caveats

- This is not a first principles talk
  - Applications first, “magazine” style
  - Read FP in Scala, Advanced Scala with Cats, etc. for theory
- Cats typeclass library
  - Scalaz examples would be similar|same

Applied FP Abstractions

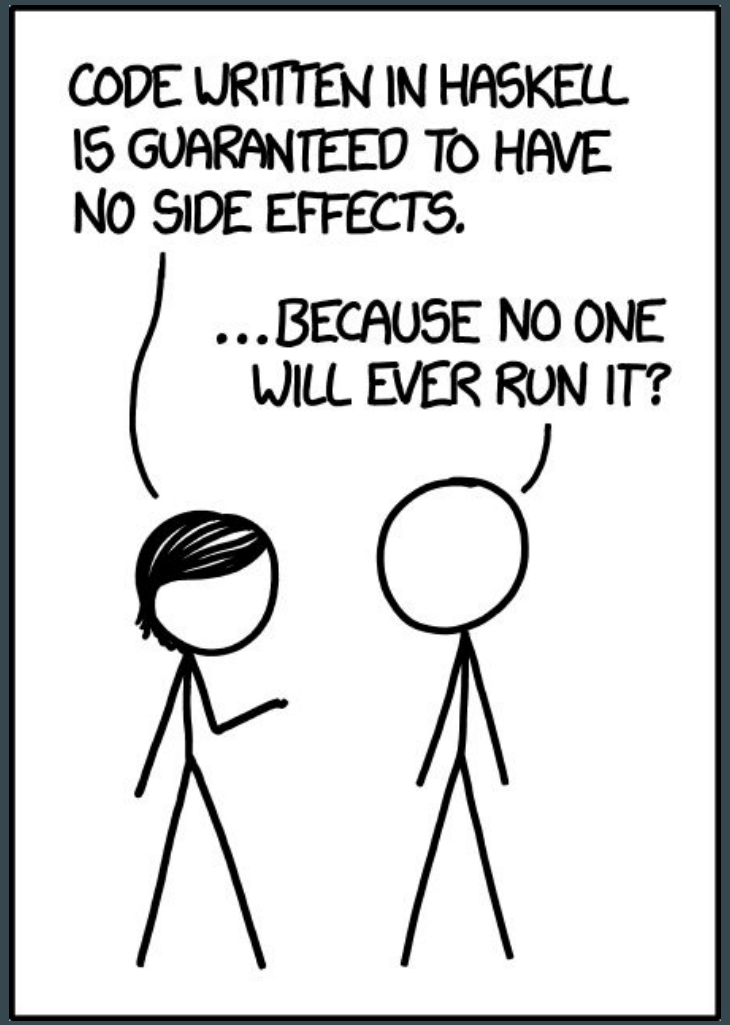


image: xkcd (Creative Commons)

Effects
vs
Side Effects

=> “A change to the world”
=> Accounted for vs unaccounted
=> Typed vs void/Unit

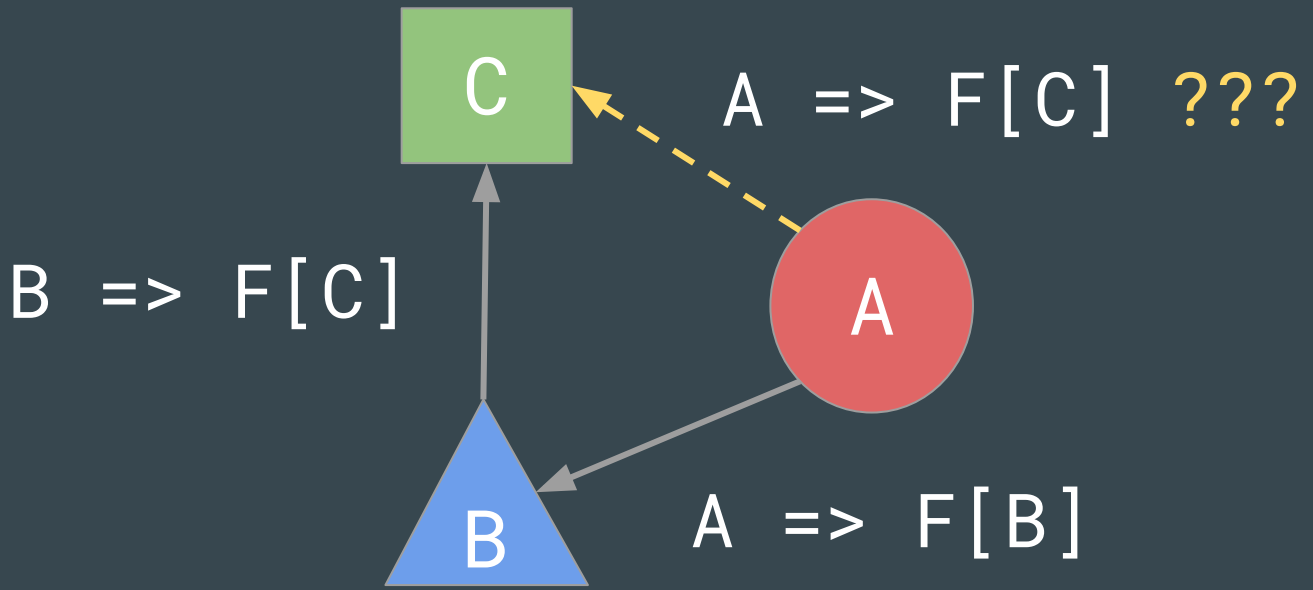
Common Computational Effects

1. **Partiality**
   a. The missing data problem
   b. Option
2. **Failure Handling**
   a. Code fails, what do you do?
   b. Either, Try
3. **IO**
   a. Interacting with external systems
   b. Stacking effects
   c. IO as glue

=> Effects are central to enterprise programming

=> Composition is the essence of functional programming

But how do we compose effects...?



Partiality

- Frequent in enterprise:
  - Empty DB/API queries
  - Function has no result
  - Object missing a field

NPE (ノ°□°)ノ

- Imperative languages use **null** to deal with this situation, *but you know what comes with it...*

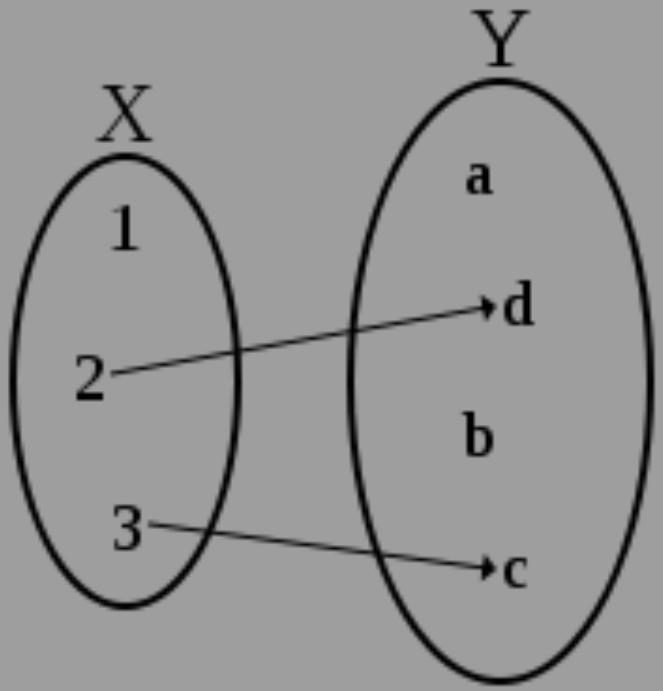


image: wikimedia foundation

Option

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](a: A) extends Option[A]
```

=> Algebraic data type or “ADT”
=> Type constructor
=> Introduced to solve the Java **null** problem
=> Simple yet very powerful tool

So we have a powerful effect abstraction for partiality.

But what are imperative people doing with it?

Using it just like Java null

(ー_ゝー)

Partiality Example 1

```
// Scala's Map implements get
// def get[K,V](k: K): Option[V]

val personLocation = Map(
  "Jerry Seinfeld"   -> "Manhattan",
  "George Costanza" -> "Manhattan",
  "Frank Costanza"   -> "Queens",
  "Elaine Benes"     -> "Brooklyn"
)
val locationTransit = Map(
  "Manhattan" -> List("J", "Z"),
  "Brooklyn"  -> List("L", "G", "K")
)
```

Partiality Example 1

```
/*
 * A simple chained cache lookup
 * 1. Select person from personLocation
 * 2. Select their subway stops from locationTransit
 * 3. Transform stops into a comma-separated string
 */
def listStops(name: String): String = ???
```

Partiality Ex. 1 (imperative)

```
def listStopsImperative(name: String): String = {
 val location = personLocation.get(name)
 var message: Option[String] = None
 if (location.isDefined) {
   val transit = locationTransit.get(location.get)
   if (transit.isDefined) {
     // Build the output string
     var output = collection.mutable.StringBuilder.newBuilder
     for (s <- transit.get) {
       output.append(s + ",")
     }
     message = Some(output.result())
   }
 }
 if (message.isDefined) message.get
 else "User or stops not found!"
}
```

Partiality Ex. 1 (functional)

```
def listStopsFunctional(name: String): String =
 personLocation.get(name)
   .flatMap(locationTransit.get)
   .fold("Not found!")(_.mkString(","))


//scala> listStopsFunctional("Jerry Seinfeld")
//res0: String = J,Z

//scala> listStopsFunctional("Newman")
//res1: String = Not found!
```

Partiality Ex. 1 (functional)

```
def listStopsSugar(name: String): Option[String] =
 for {
   location <- personLocation.get(name)
   transit  <- locationTransit.get(location)
 } yield transit.mkString(",")



def listStops(name: String): String =
  listStopsSugar(name).getOrElse("Not found!")
```

Partiality Example 2

```
// A nested Java object structure with nullable fields
class Token(val content: String)

class APIClientConfig(
  val host: String, val version: Int, val token: Token)

class JavaAPIClient(val config: APIClientConfig)
```

Partiality Example 2

```
val good: JavaAPIClient =
  new JavaAPIClient(
    new APIClientConfig("host",1,new Token("content")))

// Null field will throw NPE on access!
val bad: JavaAPIClient =
 new JavaAPIClient(new APIClientConfig("host", 1, null))


//scala> bad.config.token.content.length
//java.lang.NullPointerException
//  ... 36 elided
```

Partiality Example 2

```
// Lift the getter into Option
// We get an arrow from A => F[B]
// This is known as a Kleisli arrow
type SafeGetter[A, B] = Kleisli[Option, A, B]
def safeGetter[A, B](f: A => B): SafeGetter[A, B] =
  Kleisli(a => Option(f(a)))

implicit class GetOps[A, B, C](sg: SafeGetter[A, B]) {
  private def lift(f: B => C): B => Option[C] =
    b => Option(f(b))

  def ?(f: B => C): SafeGetter[A, C] =
    sg andThen lift(f)
}
```

Partiality Example 2

```
// Compose safe getters into an accesor function
val tokenLength: SafeGetter[JavaAPIClient, Int] =
 safeGetter[JavaAPIClient, APIClientConfig](_.config)
   ? (_.token)
   ? (_.content)
   ? (_.length)

//scala> tokenLength(bad)
//res0: Option[Int] = None

//scala> tokenLength(good)
//res1: Option[Int] = Some(7)
```

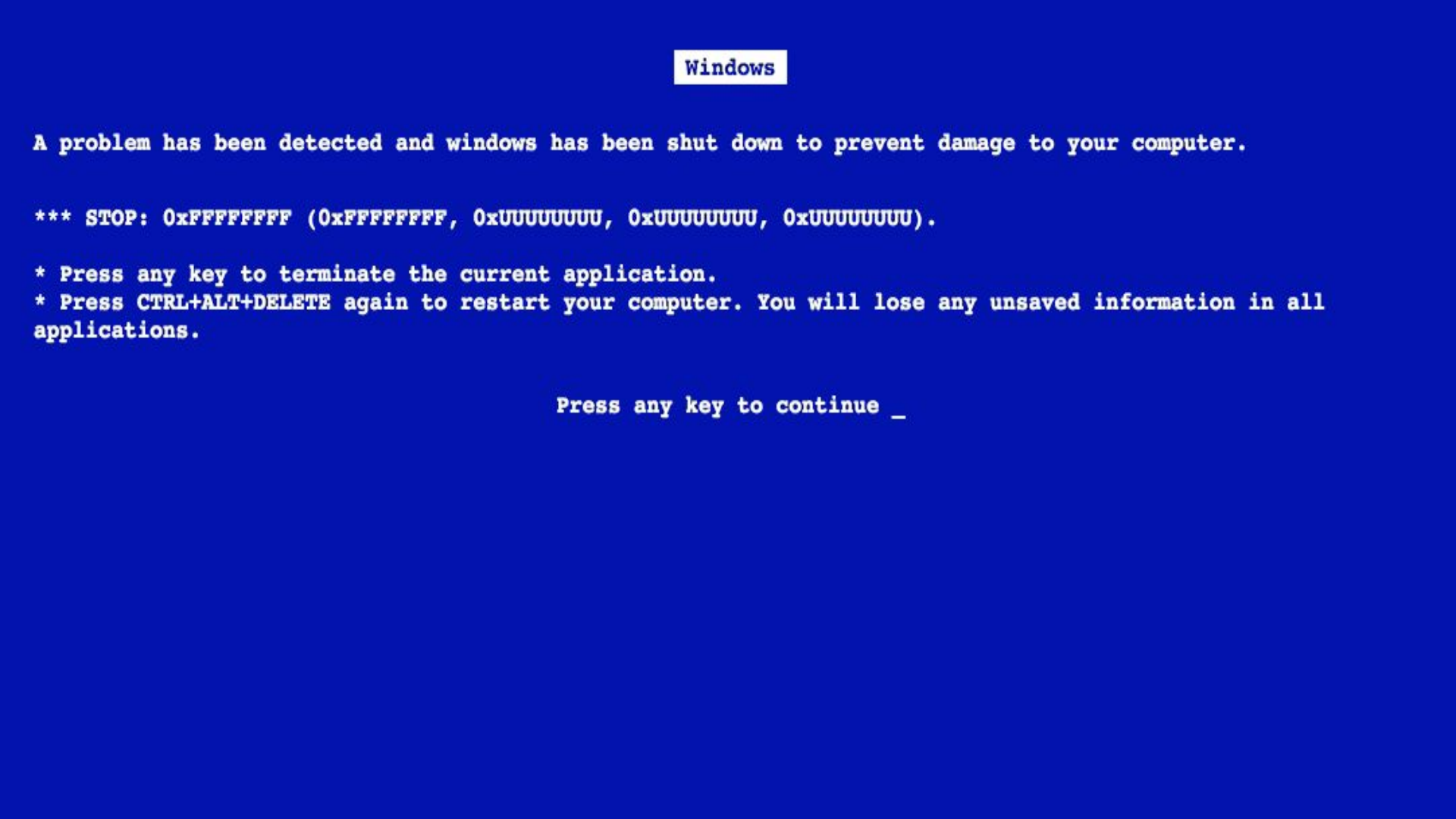
- Why does the imperative style make something so simple **complicated**?

- Lacks effect abstraction
  - Reinventing the same computational patterns every time

- Functional approach in Scala creates a declarative pipeline

The difference is in using **combinators** as they were intended…

...for *combining* effects!


Windows
A problem has been detected and windows has been shut down to prevent damage to your computer.
*** STOP: 0xFFFFFFFF (0xFFFFFFFF, 0xUUUUUUUU, 0xUUUUUUUU, 0xUUUUUUUU).
* Press any key to terminate the current application.
* Press CTRL+ALT+DELETE again to restart your computer. You will lose any unsaved information in all applications.
Press any key to continue _

Failure

- **Exceptions** naturally exist in programming
  - Enterprise people know them well… show stoppers
- Traditionally dealt with using **try-catch-finally**
  - Boilerplate
  - Hard to attribute

image: wikimedia foundation

Failure

- Scala's exceptions are typed values when not thrown!

```
// Right biased
sealed trait Either[Throwable, +A]
case class Left[A](t: Throwable) extends Either[Throwable, A]
case class Right[A](a: A) extends Either[Throwable, A]

// Constructor behaves like a try-catch block
sealed trait Try[+A]
case class Failure(t: Throwable) extends Try[Throwable]
case class Success[A](a: A) extends Try[A]
```

Failure Example

```
/*
* Modeling a website signup flow.
* `signupFlow` encrypts password and validates email
*/
def signupFlow(request: AccountRequest): AccountResponse = ???

case class AccountRequest(
  user: String, pw: String, email: String)

sealed trait AccountResponse
case class Denied(msg: String) extends AccountResponse
case class Granted(
  user: String, encryptedPW: String, validEmail: Boolean
) extends AccountResponse
```

Failure Example

```
// But our `encrypt` function has an unchecked exception
def encrypt(text: String, seed: Int): String =
  if (text.contains("mailman")) {
    throw new IllegalArgumentException("Bad input!")
  } else {
    text.map(i => (i << 1).toChar)
  }

// Unsafe call to `head` can throw NoSuchElementError
def validateEmail(email: String): Boolean = {
  val validDomains = Set("com", "org", "gov", “eu”)
  validDomains contains email.split('.').tail.head
}
```

Failure Example (imperative)

```
/*
 * The "Java" style looks simple here.
 * But can we chain together two functions that throw?
*/
def signupFlowImperative(request: AccountRequest): AccountResponse = {
  try {
    val encrypted = encrypt(request.pw, 10)
    val validEmail = validateEmail(request.email)
    Granted(request.user, encrypted, validEmail)
  } catch {
    case e: Throwable => Denied(e.getMessage)
 }
}
```

Try-Catch makes this simple enough.

But what if we wanted to chain multiple of these operations together?

Imperative version doesn't compose... ಠ∩ಠ

Failure Example (functional)

```
// We can use a type alias to bind Either's left side
type Result[A] = Either[Throwable, A]

// Then safely lift our setup functions into Result
def safeEncrypt(
  text: String, seed: Int): Result[String] =
  Either.catchNonFatal(encrypt(text, seed))

def safeValidate(email: String): Result[Boolean] =
  Either.catchNonFatal(validateEmail(email))
```

Failure Example (functional)

```
def safeSetup(req: AccountRequest): Result[AccountResponse] =
 Applicative[Result]
   .map2(safeEncrypt(req.pw, 1), safeValidate(req.email))(
     (pw, valid) => Granted(req.user, pw, valid)
   )

def safeSignup(request: AccountRequest): AccountResponse =
 safeSetup(request).fold(t => Denied(t.getMessage), identity)

//scala> safeSignup(
  AccountRequest("Newman", "mailman1999", "newman@usps.gov"))
//res0: AccountResponse = Denied(Bad input!)

//scala> safeSignup(
  AccountRequest("Jerry", "seinfeld", "jerry@comedy.com"))
//res1: AccountResponse = Granted(Jerry,æÊÒÜÌÊØÈ,true)
```

Applicative opens up “parallelism” in our code flow.

It allows us to abstract over data in separate contexts...

IO 

- Communicating with external resources is critical

- Imperative world does not have a principled way to handle IO
  - **try-catch** blocks
  - **async-await** using event loop
  - Even Scala **Future**...

Cats Effect IO

- Referentially transparent
- Composable
  - Allows the separation of actions and execution (as we will see)
- Rich control API
  - *unsafeRunSync*
  - *runAsync*
  - *attempt* (returns Either)
  - *cancelable*

IO Example

```
// Records in data lake (S3) with key relationships
case class Location(name: String)

case class Restaurant(name: String, location: Location)

case class Review(review: String)


// S3 “buckets”
val searchDatalake: mutable.Map[String, Restaurant]

val reviewsDatalake: mutable.Map[Restaurant, List[Review]]
```

IO Example

```
// We want to implement a “search and review” feature…
def searchAndReview(search: String,
  p: Restaurant => Boolean,
  review: Review) = ???

// But we’re forced to use an impure API from S3
def fetch[K, V](bucket: mutable.Map[K, V], key: K): Option[V] = {
  Thread.sleep(1000)
  bucket.get(key)
}

def write[K, V](bucket: mutable.Map[K, V], key: K, value: V): Int = {
  Thread.sleep(1000)
  bucket.put(key, value)
  1
}
```

IO Example (imperative)

```
def searchRestaurantsImperative(search: String): (Restaurant, Coordinates) = {
  var restaurant: Option[Restaurant] = None
  var coordinates: Option[Coordinates] = None
  try {
    restaurant = fetch(restaurantsDatalake, search)
    if (restaurant.isDefined) {
      coordinates = fetch(locationsDatalake, restaurant.get.location)
    }
  } catch {
    case e: Exception => println("Fetch failed somewhere... we don't know")
  }
  (restaurant.get, coordinates.get)
}
```

IO Example (functional)

```
/*
 * For convenience and reuse, we suspend the original API in IO.
 * We use OptionT in `fetch` to simplify our combinators.
*/
def ioFetch[K,V](dl: mutable.Map[K, V], key: K): OptionT[IO,V] =
  OptionT[IO,V](IO(fetch(dl, key)))


def ioWrite[K,V](bucket: mutable.Map[K, V], key: K, value: V): IO[Int] =
  IO(write(bucket, key, value))
```

IO Example (functional)

```
// Simply a matter of delegating to the fetch function
def searchRestaurants(
  search: String
): OptionT[IO, Restaurant] = ioFetch(searchDatalake, search)


// Given a restaurant and review, insert the review
def insertReview(restaurant: Restaurant, review: Review): IO[Int] =
 for {
   reviews     <- ioFetch(reviewsDatalake, restaurant).value
   insert       = reviews.fold(List(review))(review :: _)
   numInserted <- ioWrite(reviewsDatalake, restaurant, insert)
 } yield numInserted
```

IO Example (functional)

```
// Tying it all together
def searchAndReview(
  search: String,
  p: Restaurant => Boolean,
  review: Review
): IO[Int] =
  for {
    restaurant  <- searchRestaurants(search).value
    inserted    <- restaurant.fold(0.pure[IO])(
      r => if (p(r)) insertReview(r, review) else 0.pure[IO])
  } yield inserted
```

IO Example 2 (functional)

```
// A final tidbit using traverse
// Flips inner and outer layers F[_] => G[F[_]]
def insertManyReviews(
  rr: List[(Restaurant, Review)]
): IO[List[Int]] =
  rr.traverse {
    case (rest, rev) => insertReview(rest, rev)
  }
```

The IO type can be used as a "systems glue"...

Its laziness allows us to separate program definition and execution,

& improves testability!

Other cool uses of IO…

- Creating **safe** bindings over impure libraries (i.e. Kafka)
- **Composable** executor programs for use in Spark applications
- **Pluggable** effect type in database libraries

Did you notice patterns?

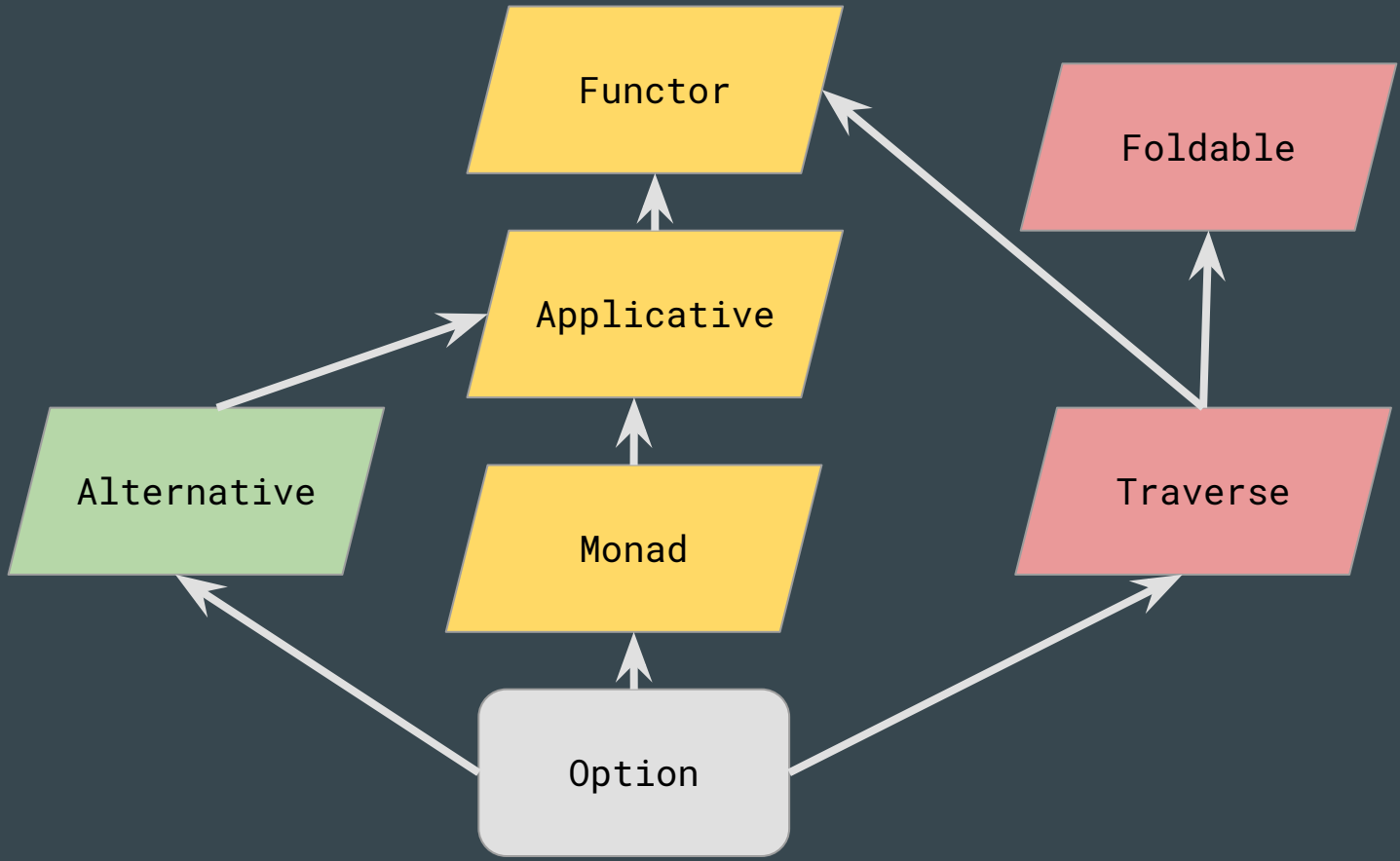
```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

trait Applicative[F[_]] extends Functor[F] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
  def pure[A](value: A): F[A]
}

trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]
}
```

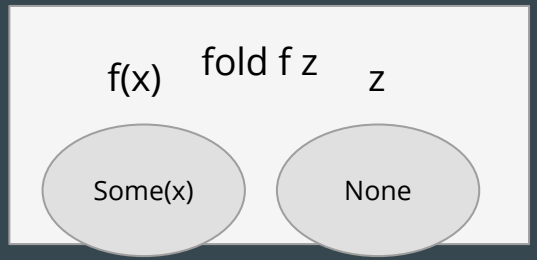
All the types we used belong to the above (and more)

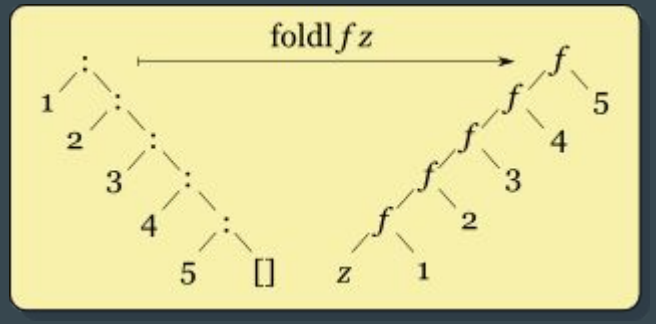
Ex: Option


Functor
Foldable
Applicative
Alternative
Traverse
Monad
Option

A quick aside about folds...

- Seen everywhere in FP
- *Foldable* is actually a typeclass in Cats and Scalaz
- Lists, sets, maps, trees can fold
- **Non-collection types are foldable too, as we have seen!**





Of course, things can get much more advanced from here…

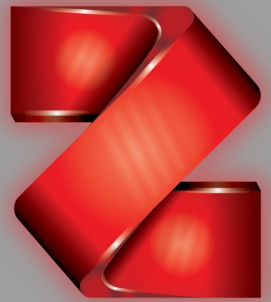
=> We want to use these patterns, but how do we introduce this stuff at work?

As a functional programmer, you must be a teacher and a salesperson at the same time.

8 Methods for Functional Programming Advocacy

8. Principled Libraries

- Very smart folks have implemented great tools for us - **let's use them**
- **Lead by example**
  - Cats
  - Scalaz/ZIO
  - Doobie
  - FS2
  - Shapeless



images: open source

7. Be a Mentor-Consultant

- ***If you're experienced in FP…***
  - Encourage people on your team who are learning FP to try it in a project
  - **Offer your time for workshops, architecture reviews, debugging**
  - **Demo your functional code and show it off!**

6. New to FP? Just Do It

- Find a small project and introduce some of these patterns
- If you only read and think about *skiing*, you will never learn to ski
- Probably best to not do this in the middle of a huge deliverable

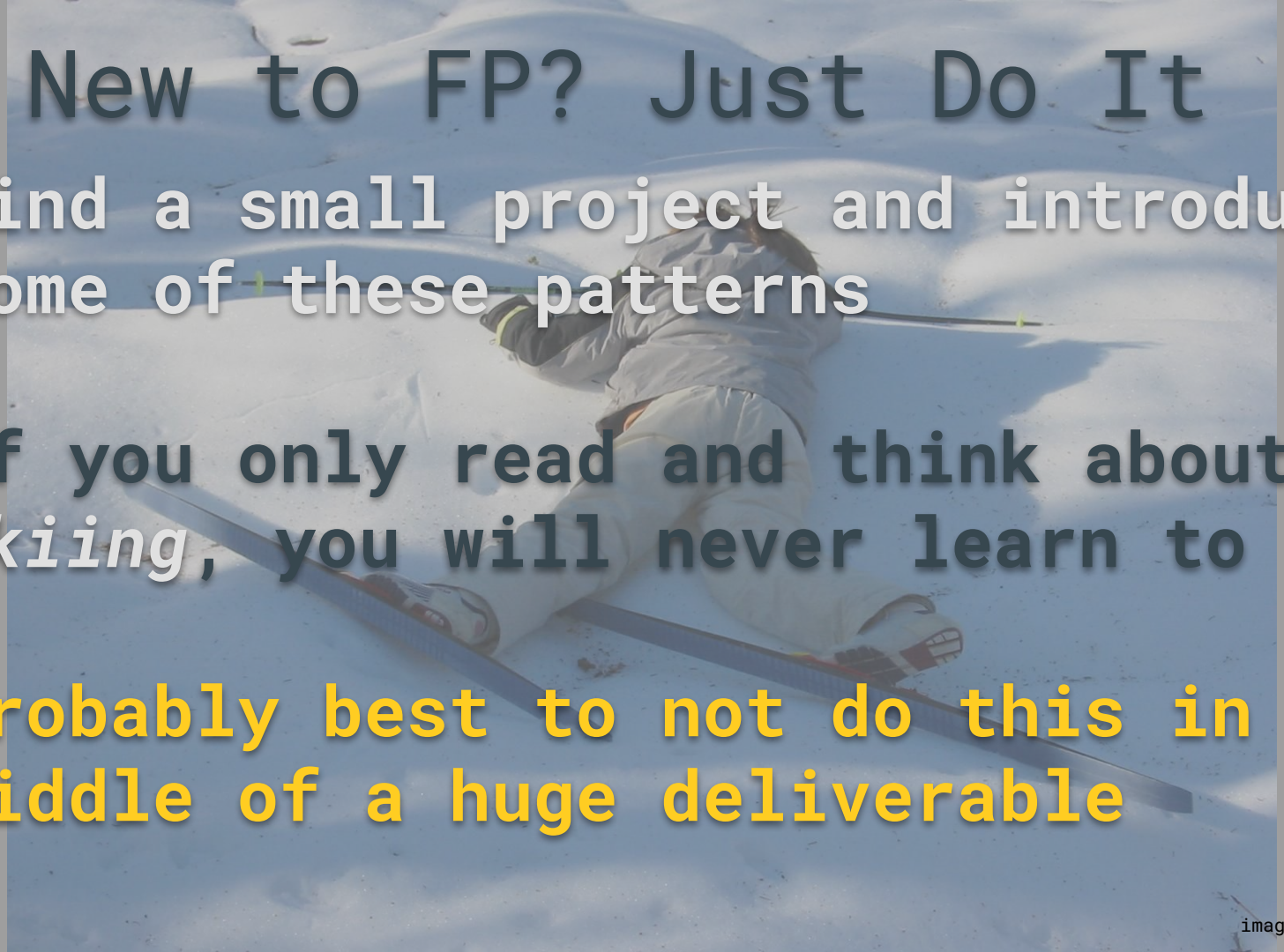


image: wikimedia

5. Focus on Business Value

- **You won't sell FP by shouting about category theory at work**

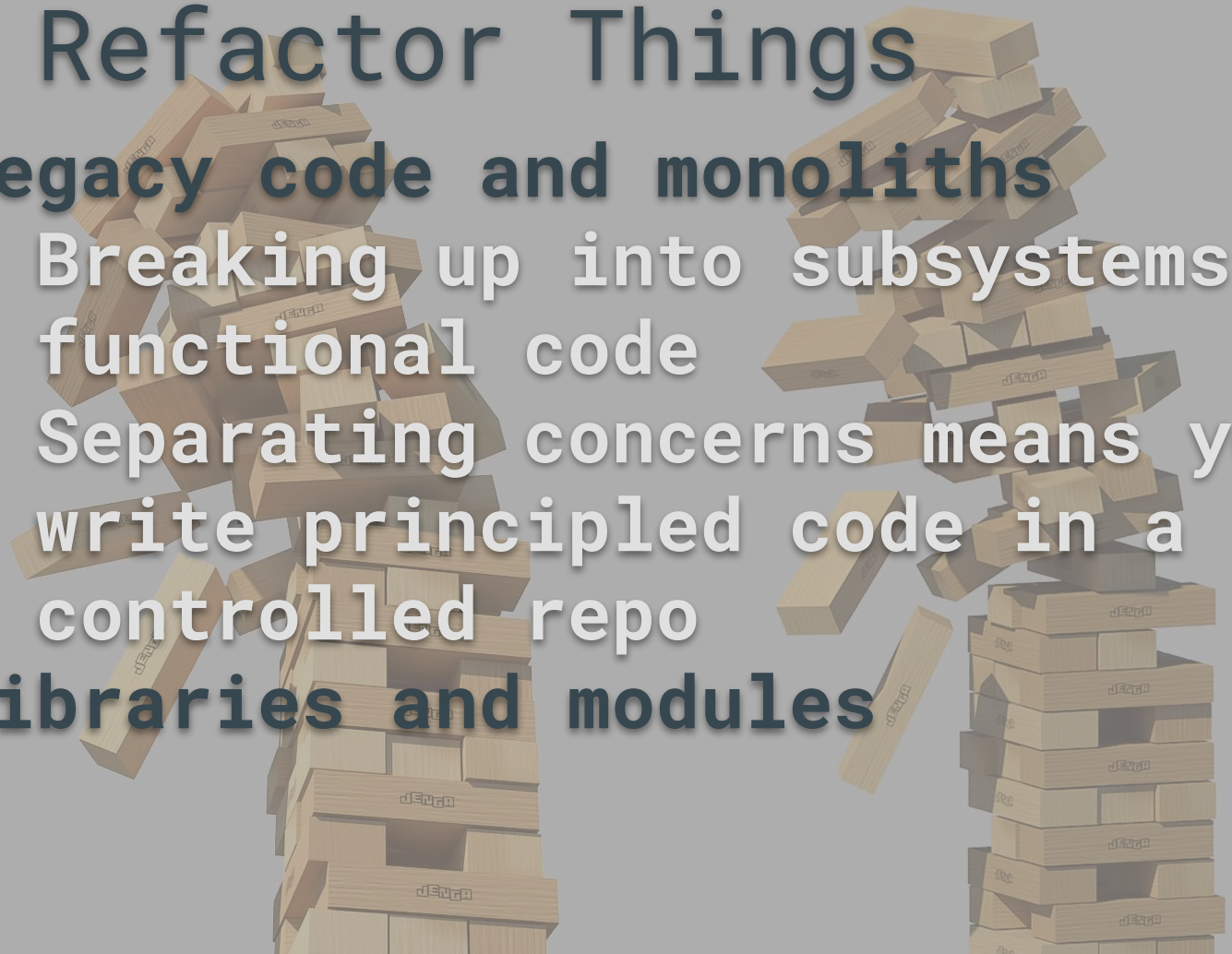
- **FP brings tangible improvements to enterprise code bases**
  - Pure functions ~ easier testing
  - Modularity ~ code reuse
  - Static types ~ correctness
  - Valued errors ~ higher uptime

4. Organize and Speak

- Set up an internal meetup or book club in office
- Submit practical talks to conferences!
- Write blog posts… people will read them.

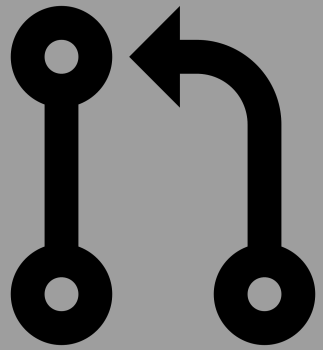
3. Refactor Things

- **Legacy code and monoliths**
  - Breaking up into subsystems with functional code
  - Separating concerns means you can write principled code in a controlled repo
- **Libraries and modules**



2. Pull Requests

- **Highly effective** and bite-sized
- **Catch impure procedural code and suggest functional alternatives**
- **Opportunity to teach and mentor other developers w.r.t functional techniques**



1. Check Ego at the Door

- Functional programming already has a bad rap for being inaccessible
- **Elitism will not get you far**
- Focus on approachability and application
  - **Save the theory for later...**

If people understand **why**, they will want to know **how**.

Thank you.

Resources...

- Slides and code examples from this talk
  - https://github.com/nasadorian/enterprise-fp/
- Rob Norris, “Functional Programming with Effects”
  - https://www.youtube.com/watch?v=po3wmq4S15A
- Bjarnason & Chiusano, “Functional Programming in Scala”
  - https://www.manning.com/books/functional-programming-in-scala
- Brian Beckman “Don’t Fear the Monad”
  - https://www.youtube.com/watch?v=ZhuHCtR3xq8
- Advanced Scala with Cats
  - https://underscore.io/books/scala-with-cats/