

# Functional Programming: *The Enterprise Edition*

...

Narek Asadorian  
LambdaConf 2018

**Slides+Code:** <https://git.io/vh4VB>

# I'm Narek.



Senior Software Engineer



@portal\_narlish



Scala & FP advocate

# My team...

# Activities Platform



salesforce

- => Streaming data, microservices & data lake
- => Millions of customer events per day
- => Spark, Storm, Akka, Kafka, Cassandra
- => **We're hiring...**

# Roll call!

Are you writing Scala at work?

Is it purely functional?

# Enterprise Functional Programming

**Oxymoron?**

# Realistic Code

- Impure
- Complicated and ugly
- Buggy
- Needs refactoring
- 10-100 normal engineers

# Ideal Code

- Pure
  - Elegant
  - Safe & provable
  - Perfect abstraction
  - Math geniuses in ivory tower
-

# Thesis:

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

Functional abstractions are well suited for commercial software.

But we will encounter *a lot* of resistance in implementation.

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



**KANYE WEST** 

@kanyewest

Follow



once again I am being attacked for  
presenting new ideas

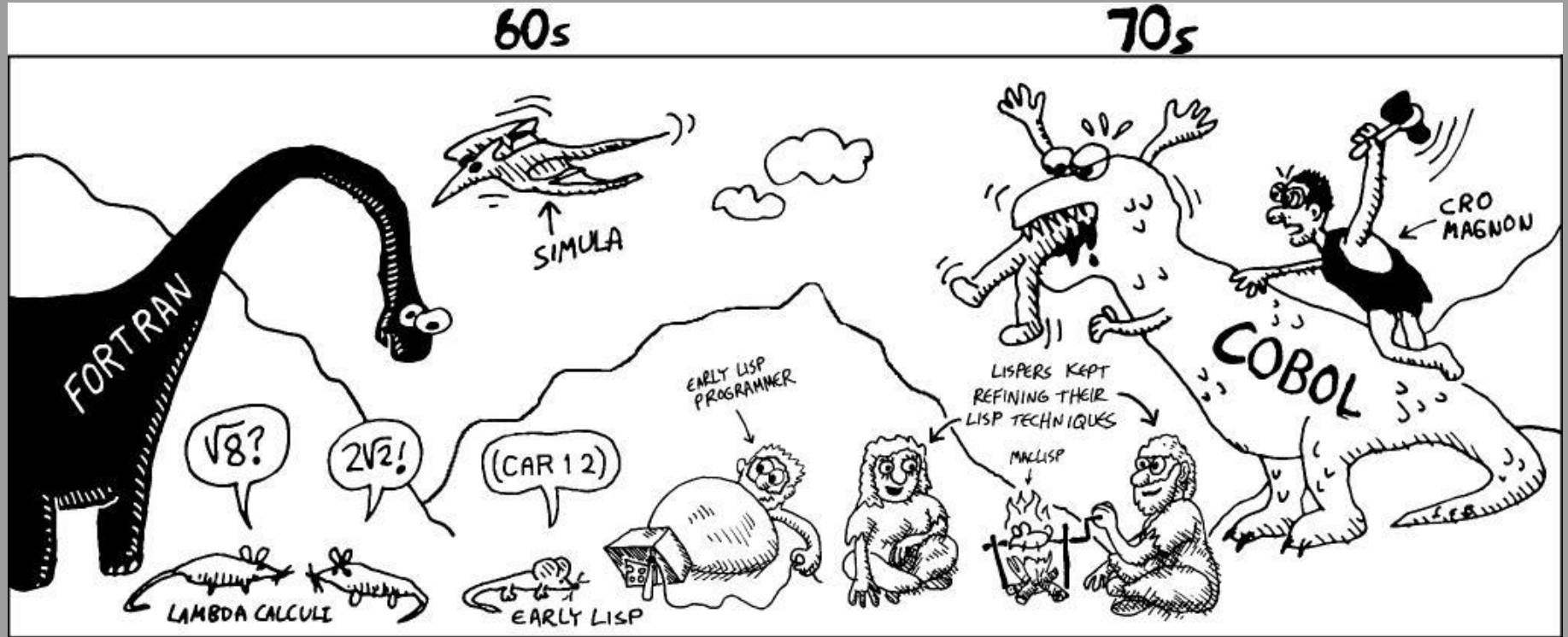
7:57 PM - 1 May 2018



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

Why tho?  
(ಠ\_ಠ)

# Why? Blame FORTRAN.



# Why? Blame universities.

- C, Java and Python taught first
- Functional concepts play a minor role (if any)
- FP langs are “too high level” to teach fundamentals with

# Why? Blame fear.

- Programmers are afraid
  - “Scary math terms”
  - 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...

# So what is this talk about?

1. Applied functional abstractions for common and problematic scenarios in enterprise programming
2. 5 strategies for introducing and fostering *functional thinking* in commercial software development

# Some Caveats

- Familiarity with Scala is assumed
  - “Java-esque” imperative examples
- This is not a first principles talk
  - Applications first
  - Read **FP in Scala** for the theory
- Cats typeclass library

# Applied FP Abstractions

CODE WRITTEN IN HASKELL  
IS GUARANTEED TO HAVE  
NO SIDE EFFECTS.

...BECAUSE NO ONE  
WILL EVER RUN IT?





# 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

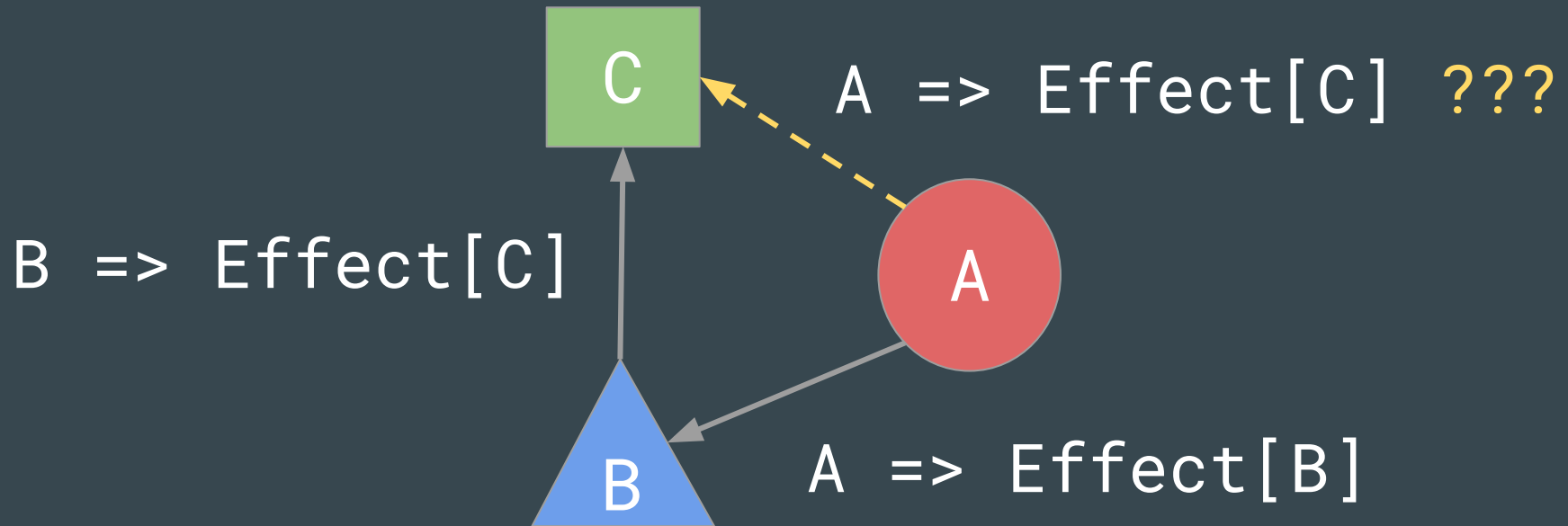
## 3. IO

- a. Interacting with the network or disk
- b. Cats IO
- c. Stacking effects

=> Computational effects are central to enterprise programming

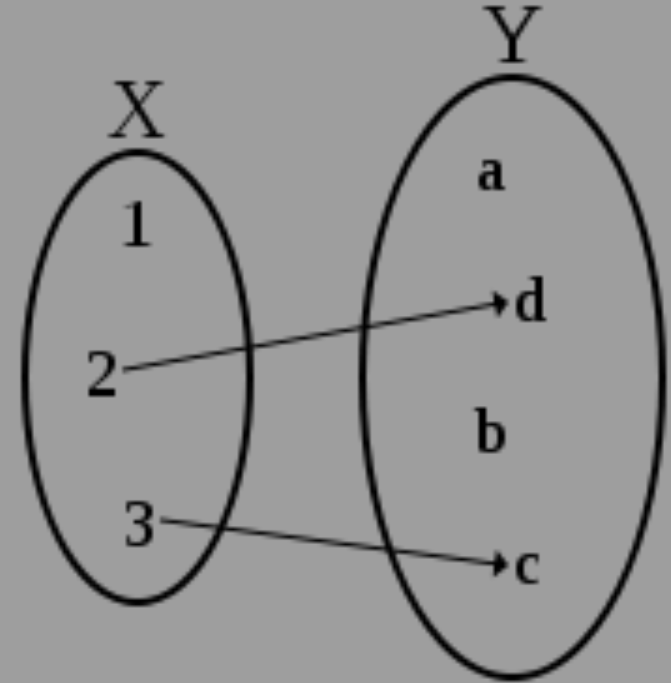
=> Composition is the essence of functional programming

# But how do we compose effects...?



# Partiality

- All values in the domain do not map to values in the codomain
- Happens frequently in enterprise:
  - Empty DB/API queries
  - Function has no result
- Imperative languages use **null** to deal with this situation, *but you know what comes with null...*



**NPE** (ノ◦□◦)ノ

# 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

Great, so we have a type constructor for partiality.

But what are imperative people doing with it?

Using it just like Java null  
(-^m)

# Partiality Example

```
// 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

```
/*  
 * A simple 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 Example (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 Example (functional)

```
def listStopsFunctional(name: String): String =  
  personLocation.get(name)  
    .flatMap(locationTransit.get)  
    .map(_._reduce(_ + " ", " " + _))  
    .getOrElse("User or stops not found!")
```

```
def listStopsFunctional2(name: String): String =  
  personLocation.get(name)  
    .flatMap(locationTransit.get)  
    .fold("User or stops not found!")(  
      _._reduce(_ + " ", " " + _))
```

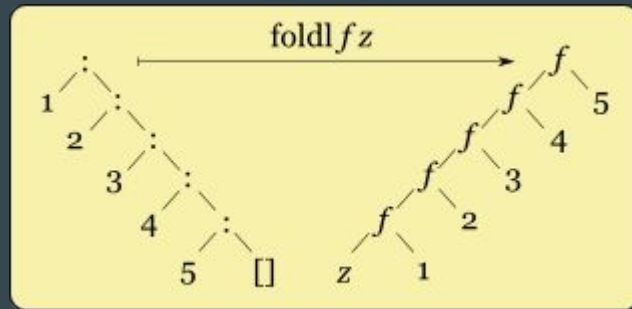
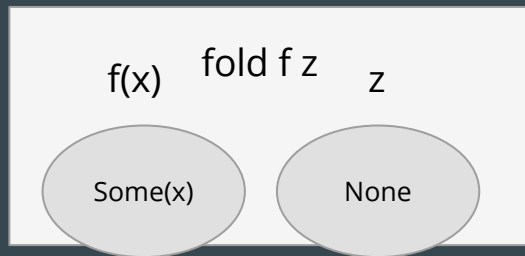
- Why does the imperative style make something so simple **complicated**?
- Lacks effect abstraction
  - Reinventing the same computational patterns every time
- Functional approach in Scala creates a **declarative pipeline**

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

...for *combining* effects!

# A quick aside about folds...

- They are 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 are seeing!**



## 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
- Traditionally dealt with using **try-catch-finally**
  - Boilerplate
  - Hard to attribute





# 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 = ???  
  
sealed 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 NoSuchElementException
def validateEmail(email: String): Boolean = {
  val validDomains = Set("com", "org", "gov")
  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...  $\emptyset \cap \emptyset$**

# Failure Example (functional)

```
// Example 1: Use Try if we want to use the offending function as-is
def signupFlowFunctional(request: AccountRequest): AccountResponse =
  Try(encrypt(request.pw, 10))
    .flatMap(encrypted =>
      Try(validateEmail(request.email))
        .map(valid => Granted(request.user, encrypted, valid)))
    .fold(t => Denied(t.getMessage), g => g)

def signupFlowFunctional2(request: AccountRequest): AccountResponse = {
  val tryGrant = for {
    encrypted    <- Try(encrypt(request.pw, 10))
    validEmail   <- Try(validateEmail(request.email))
  } yield Granted(request.user, encrypted, validEmail)

  tryGrant.fold(t => Denied(t.getMessage), g => g)
}
```

# Failure Example (functional)

```
/*  
 * Because the two `Try`s don't depend on each other  
 * we can use a weaker combinator...  
 * Look familiar? Closer to the imperative example.  
 */  
def signupFlowFunctional3(request: AccountRequest): AccountResponse = {  
  val encrypted = Try(encrypt(request.pw, 10))  
  val validated = Try(validateEmail(request.email))  
  Applicative[Try].map2(encrypted, validated)(  
    (e, v) => Granted(request.user, e, v)  
  ).fold(t => Denied(t.getMessage), g => g)  
}
```

So far we've only dealt  
with objects in our  
program...

But there's a whole world  
of data out there!



# 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 IO

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

# I0 Example

```
// Records in data lake (S3) with key relationships to each other.
sealed trait Record
case class Location(name: String) extends Record
case class Coordinates(lat: Double, lng: Double) extends Record
case class Restaurant(name: String, location: Location) extends Record

val restaurantsDataLake: mutable.Map[String, Restaurant] = mutable.Map(
  "soup"    -> Restaurant("Soup Place", Location("Queens")),
  "kabob"   -> Restaurant("Babu Bhatt's", Location("Brooklyn")),
  "diner"   -> Restaurant("Tom's Diner", Location("Manhattan"))
)

val locationsDataLake: mutable.Map[Location, Coordinates] = mutable.Map(
  Location("Queens")    -> Coordinates(40.10, 74.11),
  Location("Brooklyn")  -> Coordinates(41.12, 79.00),
  Location("Manhattan") -> Coordinates(44.00, 70.26)
)

val reviewsDataLake: mutable.Map[Restaurant, List[String]] = mutable.Map(
  Restaurant("Soup Place", Location("Queens")) -> List("Guy at counter was really rude."),
  Restaurant("Babu Bhatt's", Location("Brooklyn")) -> List("They changed the entire menu!")
)
```

# I/O Example

```
/*
 * We'll search a Restaurant by keyword and pair with Coordinates.
 * Simulating I/O operations to an S3-like data store.
 */
def searchRestaurants(search: String): (Restaurant, Coordinates)= ???

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
}
```

# I0 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 wrap the original API.  
 * Notice the use of OptionT in `fetch` due to the additional effect.  
 */  
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)

```
// Example 1: Look familiar? Notice the separation of execution & spec.
def searchRestaurantsFunctional(search: String):
  Either[Throwable, Option[(Restaurant, Coordinates)]] = {

    val io: OptionT[IO, (Restaurant, Coordinates)] = for {
      restaurant  <- ioFetch(restaurantsDataLake, search)
      coordinates <- ioFetch(locationsDataLake, restaurant.location)
    } yield (restaurant, coordinates)

    io.value.attempt.unsafeRunSync()
  }
```

# IO Example (functional)

```
// Example 2: Search a restaurant and review it based on distance.
// We can isolate the query from the last example and reuse it. Modularity!
def selectRestaurant(search: String): OptionT[IO, (Restaurant, Coordinates)] =
  for {
    restaurant <- ioFetch(restaurantsDatalake, search)
    coordinates <- ioFetch(locationsDatalake, restaurant.location)
  } yield (restaurant, coordinates)

// A new insert statement to write a review.
def insertReview(restaurant: Restaurant, review: String): IO[Int] =
  for {
    reviews      <- ioFetch(reviewsDatalake, restaurant).value
    insert        = reviews.fold(List(review))(review :: _)
    numInserted <- ioWrite(reviewsDatalake, restaurant, insert)
  } yield numInserted
```



# IO Example (functional)

```
// Bring it all together to search a restaurant then review it.
def lazyReviewer(restaurant: Restaurant, coordinates: Coordinates): IO[Int] =
  if (coordinates.lat > 40)
    insertReview(restaurant, "Terrible restaurant, too far from me!")
  else 0.pure[IO]

def findAndReview(search: String): Either[Throwable, Option[Int]] = {
  val io: OptionT[IO, Int] = for {
    (restaurant, coordinates) <- selectRestaurant(search)
    reviewsInserted <- OptionT.liftF(lazyReviewer(restaurant, coordinates))
  } yield reviewsInserted

  io.value.attempt.unsafeRunSync()
}
```

# Noticing a pattern?

```
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)

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

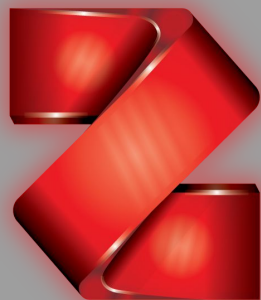
=> But how do we teach this stuff and introduce it at work?

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

# 5 Methods for Functional Programming Advocacy

# 5. Principled Libraries

- Very smart folks have implemented great tools for us - **let's use them**
- **Lead by example**
- In the Scala world...
  - Cats
  - Scalaz
  - Doobie
  - FS2
  - Shapeless



## 4. Organize and Speak

- Set up an **internal** meetup or book club
  - **Submit practical** talks to conferences!
  - **Write blog posts!**
- Communication works.**



# 3. Refactors

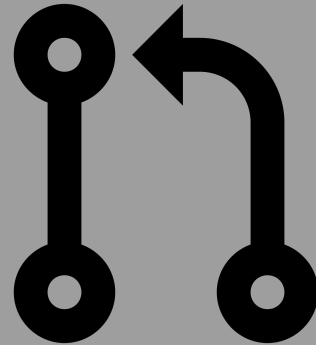
- **Legacy code & monoliths**

- 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 before it goes into staging
- 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
  - **Don't make it worse**
- 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/>
- Try not a Monad
  - <https://gist.github.com/ms-tg/6222775>