# Solving The Dependency Injection Problem With ZIO

# Agenda

- **Introduction to ZIO**
- The Dependency Injection Problem
- ZIO's Environment Type
- Using Layers to Build Environments

# Mental Model

```
type ZIO[-R, +E, +A] = R => Either[E, A]
```

- A `ZIO[R, E, A]` is a description of a computation that requires an environment `R` and may either fail with an error `E` or succeed with a value `A`
- A description of a potentially asynchronous computation
- You can think of `ZIO` as a future on steroids

# Hello ZIO

```scala
object HelloZIO extends App {

  def run(args: List[String]): ZIO[ZEnv, Nothing, Int] =
    sayHello.as(0)

  val sayHello: ZIO[Console, Nothing, Unit] =
    console.putStrLn("Hello, ZIO!")
}
```

# Composable Descriptions

A ZIO is a description of a computation instead of a running computation so they are incredibly composable:

```scala
val dice: ZIO[Random, Nothing, Int] =
  random.nextInt(6).map(_ + 1)

val pair: ZIO[Random, Nothing, Int] =
  dice.zipWith(dice)(_ + _)

val sample: ZIO[Console with Random, Nothing, Unit] =
  ZIO.collectAll_ {
    List.fill(100)(pair.flatMap(n => console.putStrLn(n.toString)))
  }
```

# Lossless Error Model

Polymorphic error type allows you to tell how your computations can fail, if they can fail at all:

```
sealed trait PasswordError
case object IOError         extends PasswordError
case object InvalidPassword extends PasswordError

val getInput: ZIO[Console, IOError, String] = ???

def validateInput(password: String): ZIO[Any, InvalidPassword, Password] = ???

val getAndValidateInput: ZIO[Console, PasswordError, Password] =
  getInput.flatMap(validateInput)

val getPassword: ZIO[Console, Nothing, Password] =
  getAndValidateInput.orElse(getPassword)
```

# Fiber Based Concurrency

Fibers are lightweight equivalents of threads:

- Can have hundreds of thousands of fibers at a time
- Semantically block but never block underlying operating system thread
- Safely interruptible
- Automatic supervision

# Parallelism

ZIO makes it simple to write safe parallel programs:

```
def decrypt(data: Array[Byte]): ZIO[Any, Nothing, String]     = ???
def validate(data: Array[Byte]): ZIO[Any, InvalidData, Unit] = ???

for {
  fiber  <- decrypt(data).fork
  _      <- validate(data)
  result <- fiber.join
} yield result
```

# Concurrency

Combinators provide strong guarantees that make it easy to reason about concurrent programs:

- *Race* - the slower of two computations will be immediately interrupted
- *Bracket* - resource will always be released when computation is terminated
- *Lock* - computation will always be executed on specified executor

# Safe Resource Usage

```
try {
    resource = acquire
    doSomething(resource)
} finally {
   resource.release
}
```

Try finally doesn't work with asynchronous or concurrent code

# Composable Resources

```
val resource1 = ZManaged[Any, Nothing, A]
val resource2 = ZManaged[Any, Nothing, B]
val resource3 = ZManaged[Any, Nothing, C]

for {
  a      <- resource1
  (b, c) <- resource2.zipPar(resource3)
} yield f(a, b, c)
```

Resources are guaranteed to be acquired in correct order and released as soon as possible

# Concurrent Data Structures

"Batteries included" library of concurrent data structures to solve any problem:

- *Ref* - concurrent state
- *Promise* - single element coordination between fibers
- *Queue* - multiple element coordination between fibers
- *STM* - full software transactional memory library

# Agenda

- Introduction to ZIO
- **The Dependency Injection Problem**
- ZIO's Environment Type
- Using Layers to Build Environments

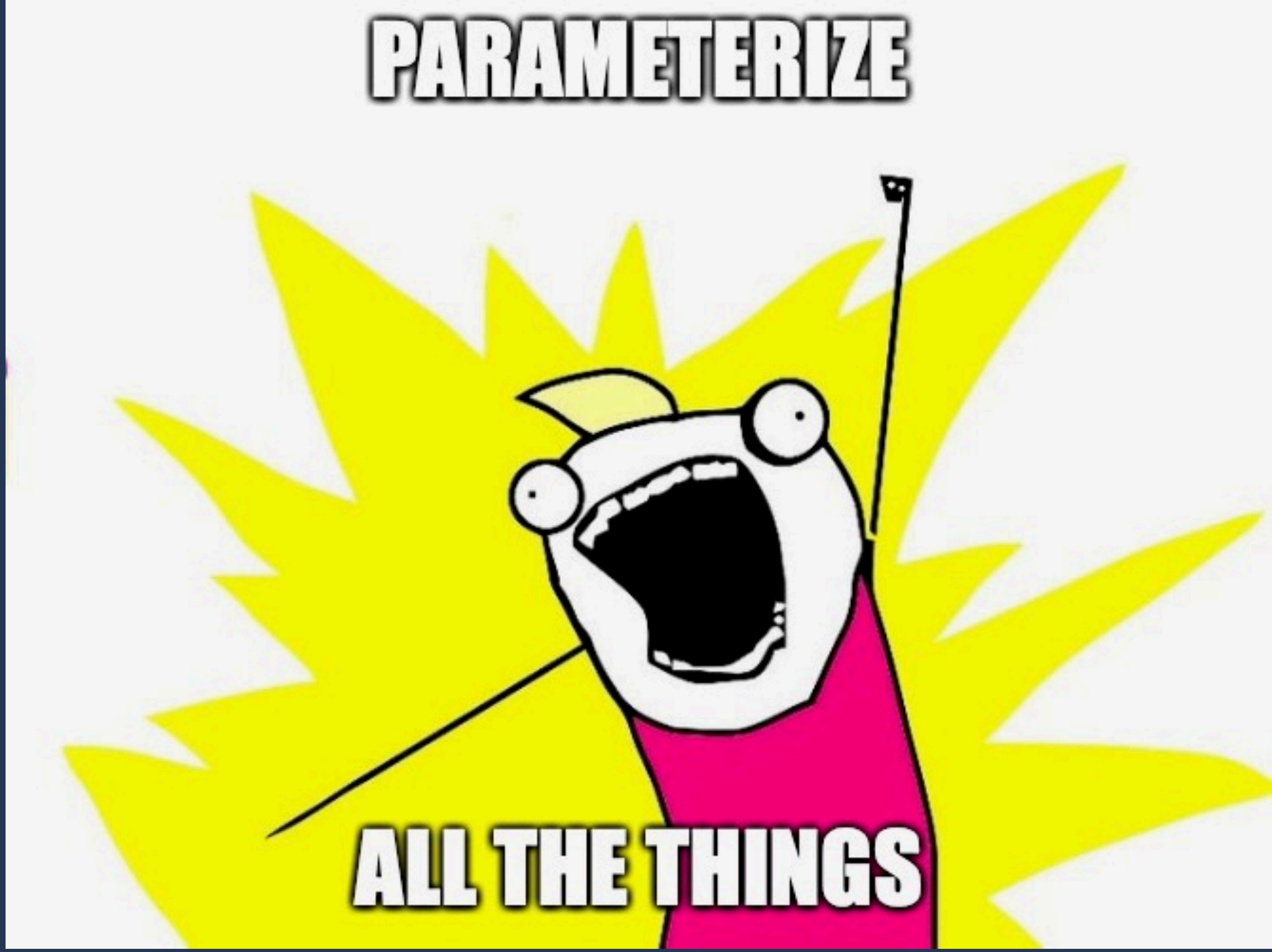
# How Do We Get Dependencies To Where They Are Needed?

```
trait Bloomberg {
  def getStockPrice(ticker: String): Future[Double]
}

def executeTrades(bloomberg: Bloomberg): Unit =
  ???

// ten layers down...

def appleStockPrice: Future[Double] =
  bloomberg.getStockPrice("AAPL")
```

# Traditional Solutions

1. Argument Passing
2. Implicit Parameters
3. Constructor Arguments
4. Cake Pattern
5. Specialized Frameworks

# Argument Passing




"Simplicitly: Foundations and Applications of Implicit Function Types" by Martin Odersky

# Argument Passing

Problems:

- Have to include in all function definitions that directly or indirectly use dependency
- Have to include in all calls to those functions
- Doesn't scale to many dependencies

# Implicit Parameters

Just make them implicit:

```
def getStockPrices(
  tickers: List[String]
)(implicit bloomberg: Bloomberg): Future[List[Double]] =
  Future.traverse(tickers)(bloomberg.getStockPrice)
```

# Implicit Parameters

Problems:

- Have to include in all function definitions that directly or indirectly use dependency
- ~~Have to include in all calls to those functions~~
- Doesn't scale to many dependencies
- Harder to reason about and more error prone

# Constructor Arguments

```
class StockTradingService(bloomberg: Bloomberg) {
  def getStockPrice(ticker: String): Future[Double] =
    bloomberg.getStockPrice(ticker)
  // more methods here
}
```

# Problems

- Allocation and deallocation of resources
- Coarse expression of dependencies
- Encourages use of dependency injection framework
- Hard to refactor

# Cake Pattern

```
trait BloombergService {
  def getStockPrice(ticker: String): Future[Double]
}

trait Bloomberg {
  def bloomberg: BloombergService
}

object LiveBloomberg extends Bloomberg {
  ???
}

trait TradeExecution { self: Bloomberg =>
  val appleStockPrice = bloomberg.getStockPrice("AAPL")
}

object LiveTradeExecution extends TradeExecution with LiveBloomberg
```

# Cake Pattern

Problems:

- Allocation and deallocation of resources
- Coarse expression of dependencies
- Boilerplate
- Hard to refactor

# Specialized Frameworks

Use macros or reflection to dynamically inject dependencies:

- Guice
- Spring
- MacWire
- Airframe
- distage

# Specialized Frameworks

Problems:

- Have to learn domain specific language
- "Magic" - when it works it works, otherwise good luck
- Lack of type safety

# Agenda

- Introduction to ZIO
- The Dependency Injection Problem
- **ZIO's Environment Type**
- Using Layers to Build Environments

# Mental Model

`ZIO[-R, +E, +A] = R => Either[E, A]`

- A `ZIO` computation can require an environment `R` to be run
- Allows us to describe our whole program and its dependencies, deferring providing their implementations until the "end of the world"
- Computations that don't require any environment have `R` of `Any`

# Accessing the Environment

Access the environment using `ZIO.environment`:

```
def getStockPrice(ticker: String): ZIO[Bloomberg, BloombergError, Double] =
  ZIO.environment[Bloomberg].flatMap(_.getStockPrice(ticker))
```

A computation that requires a Bloomberg service and either succeeds with a stock price or fails with a domain specific error

# Providing the Environment

Provide the environment using ZIO#provide:

```
val appleStockPrice: ZIO[Any, BloombergError, Double] =
  getStockPrice("AAPL").provide(Bloomberg.Live)
```

A computation that has been provided with its required environment and is ready to be run

# Environmental Requirements Compose

Combining two computations requires both of their environments:

```
def getStockPrice(ticker: Ticker): ZIO[Bloomberg, BloombergError, Double]        = ???
def logPrice(ticker: String, price: Double): ZIO[Logging, Nothing, Unit] = ???

def getAndLogPrice(ticker: String): ZIO[Bloomberg with Logging, BloombergError, Double] =
  for {
    price <- getStockPrice(ticker)
    _     <- logPrice(ticker, price)
  } yield price
```

# Environmental Requirements Scale

```
type Trading = Bloomberg with Logging

def getAndLogPrice(ticker: String): ZIO[Trading, BloombergError, Double] =
  ???
```

We can use `with` to concisely describe complex environments

# Pain Points

While a giant step forward, there have been some limitations in working with environment in previous ZIO versions:

- Boilerplate in constructing environments
- Difficulty locally modifying and eliminating environments

# Boilerplate in Constructing Environments

```
trait Bloomberg {
  def bloomberg: Bloomberg.Service
}

object Bloomberg {
  trait Service {
    def getStockPrice(ticker: String): IO[BloombergError, Double]
  }
}

trait Logging {
  def logging: Logging.Service
}

object Logging {
  trait Service {
    def logLine(line: String): UIO[Unit]
  }
}
```

# Boilerplate in Constructing Environments

```
val appleStockPrice: ZIO[Bloomberg, BloombergError, Double] =
  getAndLogPrice("AAPL").provideSome[Bloomberg] { env =>
    new Bloomberg with Logging {
      val bloomberg = env.bloomberg
      val logging = Logging.Live.logging
    }
  }
```

# Difficulty Locally Modifying and Eliminating Environments

```
def modifyClock[R <: Clock, E, A](
  zio: ZIO[R, E, A]
)(f: Clock.Service => Clock.Service): ZIO[R, E, A] =
  zio.provideSome[R] { r =>
    ???
  }
```

# The Mix Problem

```
def mix[A, B](a: A, b: B): A with B =
  ???
```

- Previous solutions, including the `Mix` type class from the ZIO Macros project and the `Enrich` type class from Netflix's Polynote project have used macros to solve this problem
- Not ideal experience for a core library feature

# Agenda

- Introduction to ZIO
- The Dependency Injection Problem
- ZIO's Environment Type
- **Using Layers to Build Environments**

# Has

`Has[A] with Has[B] with Has[C]`

- Backed by heterogenous map from types to values
- We know how to build and combine maps, we don't know how to build and combine abstract types

# Heterogenous Map

```
Has[Blocking.Service] with Has[Random.Service]
  with Has[Clock.Service] with Has[Console.Service]
  with Has[System.Service]

Map(
  Tagged[Blocking.Service] -> BlockingServiceLive,
  Tagged[Random.Service]   -> RandomServiceLive,
  Tagged[Clock.Service]    -> ClockServiceLive,
  Tagged[Console.Service]  -> ConsoleServiceLive,
  Tagged[System.Service]   -> SystemServiceLive
)
```

# Defining Services

```
type Bloomberg = Has[Bloomberg.Service]
type Logging   = Has[Logging.Service]

object Bloomberg {
  trait Service {
    def getStockPrice(ticker: String): IO[BloombergError, Double]
  }
}

object Logging {
  trait Service {
    def logLine(line: String): UIO[Unit]
  }
}
```

# ZLayer

`ZLayer[-RIn, +E, +ROut]`

- A `ZLayer[RIn, E, ROut]` is a "recipe" for building an environment of type `ROut`
- Layers can require allocation and deallocation
- Layers are shared by default, so if the same layer is used in multiple parts of your dependency graph it will only be created once

# Layers Compose

Layers compose horizontally and vertically:

- *Horizontally* - Combine two layers with ++ to get a new layer that requires both of their inputs and produces both their outputs
- *Vertically* - Combine one layer that depends on another with >>> to get a new layer that takes the input to the first and produces the output of the second

# Horizontal Composition

```
object ZEnv {
  val live: ZLayer[Any, Nothing, Clock with Console with Random with System] =
    Clock.live ++ Console.live ++ Random.live ++ System.live
}
```

# Vertical Composition

```scala
object TestEnvironment {
  val live =
    Annotations.live ++
      (Live.default >>> TestClock.default) ++
      TestConsole.default ++
      Live.default ++
      TestRandom.random ++
      Sized.live(100) ++
      TestSystem.default
}
```

# Layers Can Require Finalization

```
val bloombergLayer: ZLayer[Any, Nothing, Bloomberg] =
  ZLayer.fromManaged {
    Managed.make(acquireBloombergService)(releaseBloombergService)
  }
```

# Locally Eliminating Dependencies

```
val getAndLogPrice: ZIO[Bloomberg with Logging, BloombergError, Double] =
  ???

val liveLogging: ZLayer[Any, Nothing, Logging] = ???

val appleStockPrice: ZIO[Bloomberg, BloombergError, Double] =
  getAndLogPrice.provideSomeLayer[Bloomberg](liveLogging)
```

# Locally Updating Dependencies

Easy to locally update services:

```
val tradingLayer: ZLayer[Any, Nothing, Bloomberg with WorldBank with Logging] =
  ???

val updated = tradingLayer.updated[Logging] { loggingService =>
  // update logging service to display in black and white
  ???
}
```

# Locally Overriding Dependencies

Easy to locally override services:

```
val tradingLayer: ZLayer[Any, Nothing, Bloomberg with WorldBank with Logging] =
  ???

// temporarily log data to console for debugging purposes
val consoleLogging: ZLayer[Any, Nothing, Logging] = ???

// ++ overrides existing service implementation
val updated = tradingLayer ++ consoleLogging
```

# Impossible to Create Cyclic Dependencies

```
lazy val chickenLayer: ZLayer[Egg, Nothing, Chicken] = ???
lazy val eggLayer: ZLayer[Chicken, Nothing, Egg]     = ???

lazy val circular: ZLayer[Chicken, Nothing, Egg] =
  eggLayer >>> chickenLayer >>> eggLayer
```

# Benefits Of ZLayer

- Powerful
- Concise and ergonomic
- Type safe
- Resource safe
- No "magic"

# Conclusion

- ZIO is here to help you solve complex problems
- ZIO's environment type provides a comprehensive solution to the dependency injection problem
- Try it out and visit us on Github and Discord

# Thank You

- Maxim Schuwalow and Piotr Golebiewski for their pioneering work on ZIO Macros and heterogenous maps
- Septimal Mind for implementation of type tagging
- John De Goes for his mentorship and leadership
- All of the users who tried earlier versions of ZIO and gave us honest feedback on what worked and what didn't
- All of you for taking time out of your day to attend