

# Practical Functional Programming

# Pure I/O

## in Scala

# Previously, **Immutability**

# Immutable Class

```
case class Customer(id: Int, firstName: String, lastName: String)

// Create an new instance
val customer = Customer(id = 1, firstName = "John", lastName = "Doe")
val name = customer.firstName

// Create a modified copy of an instance
val modifiedCustomer = customer.copy(lastName = "Martin")
// `customer` remains unmodified

// Compare instances by value
val sameCustomer = Customer(id = 1, firstName = "John", lastName = "Doe")
assert(customer == sameCustomer)
```

# Expressions

```
val status = if enabled then "On" else "Off" // `if` expression
```

```
val mark = color match { // `match` expression
  case Red => 2
  case Orange => 4
  case Green => 6
}
```

```
val altitude = { // { ... } expression
  val y = slope * t

  if y < -threshold then -threshold
  else if y > threshold then threshold
  else y
}
```

# Simple Immutable enum

```
enum Direction {  
  case North, South, West, East  
}  
  
case class Position(x: Int, y: Int) {  
  def move(direction: Direction): Position =  
    direction match {  
      case North => this.copy(y = this.y - 1)  
      case South => this.copy(y = this.y + 1)  
      case West => this.copy(x = this.x - 1)  
      case East => this.copy(x = this.x + 1)  
    }  
}
```

# Immutable enum on steroids

```
enum Action { // ADT (Algebraic Data Type)
  case Sleep
  case Walk(direction: Direction)
  case Jump(position: Position)
}

case class Player(position: Position) {
  def act(action: Action): Player =
    action match { // Pattern Matching
      case Sleep => this
      case Walk(direction) => Player(position.move(direction))
      case Jump(position) => Player(position)
    }
}
```

Previously, **functional programming**





# Functional Programming

- Programming with **functions** that are
  - **Deterministic**: same arguments implies same result
  - **Total**: result always available for arguments, no exception
  - **Pure**: no side-effects, only effect is computing result
- And **values** that are **immutable**



# Refactorings Break Impure Programs

# Referential Transparency

- A benefit of FP is **referential transparency**.
- **Typical refactorings cannot break a working program** 👍.
- Applies to the following refactorings:
  -  **Extract Variable**
  -  **Inline Variable**
  -  **Extract Method**
  -  **Inline Method**

# Console Operations

```
object Console {  
  def printLine(o: Any): Unit = println(o)  
  def readLine(): String = StdIn.readLine()  
}
```

# A Working Program

```
object ConsoleApp {  
  def main(args: Array[String]): Unit = {  
    Console.println("What's player 1 name?")  
    val player1 = Console.readLine()  
    Console.println("What's player 2 name?")  
    val player2 = Console.readLine();  
    Console.println(s"Players are $player1 and $player2.")  
  }  
}
```

What's player 1 name?

> Paul

What's player 2 name?

> Mary

Players are Paul and Mary.

# Broken Extract Variable Refactoring

```
object ConsoleApp {  
  def main(args: Array[String]): Unit = {  
    val s = Console.readLine()  
    Console.println("What's player 1 name?")  
    val player1 = s  
    Console.println("What's player 2 name?")  
    val player2 = s;  
    Console.println(s"Players are $player1 and $player2.")  
  }  
}
```

> Paul

What's player 1 name?

What's player 2 name?

Players are Paul and Paul.

# Broken Inline Variable Refactoring

```
object ConsoleApp {  
  def main(args: Array[String]): Unit = {  
    Console.println("What's player 1 name?")  
    Console.println("What's player 2 name?")  
    val player2 = Console.readLine();  
    Console.println(s"Players are ${Console.readLine()} and $player2.")  
  }  
}
```

What's player 1 name?

What's player 2 name?

> Paul

> Mary

Players are Mary and Paul.

# Building a Pure Program from the Ground Up

# Describing a Program

```
class IO[+A]
```

- Describes a **program** performing I/Os
- When run, will eventually yield a **result** of type A
- Simplified, don't handle errors or exceptions



# Infallible Program as Immutable Value

```
object IO {  
  def succeed[A](a: => A): IO[A] = IO(() => a) // ...  
  // Wraps a presumably infallible expression  
  // `=> A` is equivalent to `() => A`  
  // But `succeed` should be called with `IO.succeed(expr)`  
  // `expr` argument is only evaluated when `a` parameter is used  
}
```

- Simple immutable class
- Holds a parameterless **side-effecting** function

# Elementary Console Programs

```
object Console {  
  def printLine(o: Any): IO[Unit] =  
    IO.succeed(/* () => */ println(o))  
  
  val readLine: IO[String] =  
    IO.succeed(/* () => */ StdIn.readLine())  
}
```

# A Value Containing Void (`Unit`)

- `Unit` is a class with only 1 instance written as `()`
- `()` is an immutable **value** that bears no information
- Somehow an empty tuple

# Chaining Programs

```
case class IO[+A](unsafeIO: () => A) { ioA => // Make `ioA` an alias for `this`  
  // Could be called `thenChain`  
  def flatMap[B](cont: A => IO[B]): IO[B] = {  
    IO(  
      () => {  
        val a: A = ioA.unsafeIO()  
        val ioB: IO[B] = cont(a)  
        val b: B = ioB.unsafeIO()  
        b  
      }  
    )  
  } // ...  
}
```

# Transforming Result of Program

```
case class IO[+A](unsafeIO: () => A) { ioA => // ...  
  // Could be called `thenTransform`  
  def map[B](trans: A => B): IO[B] = {  
    IO(  
      () => {  
        val a: A = ioA.unsafeIO()  
        val b: B = trans(a)  
        b  
      }  
    )  
  }  
}
```

# Instantiating a Program

```
object ConsoleApp {  
  val helloApp: IO[Unit] =  
    Console.println("What's your name?").flatMap { _ =>  
      Console.readLine.flatMap { name =>  
        Console.println(s"Hello $name!")  
      }  
    }  
}  
  
def main(args: Array[String]): Unit = {  
  val program = helloApp  
}  
}
```

# But Program Does Not Run 😲

```
object ConsoleApp {  
  // ...  
  def main(args: Array[String]): Unit = {  
    val program = helloApp  
    println(program)  
  }  
}
```

- Will print something like `IO(io.pure.IO$$Lambda$19/0x0000000800098c40@42eca56e)`
- This is just an immutable **value**, it performs no side-effect, it's **pure** 😇.
- Need an **interpreter** to run!

# Interpreting a Program

```
object Runtime {  
  def unsafeRun[A](io: IO[A]): A = io.unsafeIO()  
}
```



# Running a Program

```
object ConsoleApp {  
  // ...  
  
  def main(args: Array[String]): Unit = {  
    val program = helloApp  
    // PURE-only above ^^^^^ (programs are values)  
    Runtime.unsafeRun(program) // Only this line is IMPURE!!!  
  }  
}
```

- Sure, unsafeRun call point (***edge of the world***) is **impure** 😈...
- But the **rest of the code** is fully **pure** 😇!

# for Comprehension

# Elementary Random Programs

```
object Random {  
  def nextIntBetween(min: Int, max: Int): IO[Int] =  
    IO.succeed(scala.util.Random.nextInt(max - min + 1) + min)  
}
```

# Pyramid of maps and flatMaps 🍆

```
val welcomeNewPlayer: IO[Unit] =  
  Console.println("What's your name?").flatMap { _ =>  
    Console.readLine.flatMap { name =>  
      Random.nextIntBetween(0, 20).flatMap { x =>  
        Random.nextIntBetween(0, 20).flatMap { y =>  
          Random.nextIntBetween(0, 20).flatMap { z =>  
            Console.println(s"Welcome $name, you start at coordinates ($x, $y, $z).")  
          }  
        }  
      }  
    }  
  }
```

# Flatten Them All 🙇

```
val welcomeNewPlayer: IO[Unit] =  
  for {  
    _ <- Console.println("What's your name?")  
    name <- Console.readLine  
    x <- Random.nextIntBetween(0, 20)  
    y <- Random.nextIntBetween(0, 20)  
    z <- Random.nextIntBetween(0, 20)  
    _ <- Console.println(s"Welcome $name, you start at coordinates ($x, $y, $z).")  
  } yield ()
```

Syntactic sugar that calls flatMap and map

# Intermediary Variable

```
val printRandomPoint: IO[Unit] =  
  for {  
    x <- Random.nextIntBetween(0, 20)  
    y <- Random.nextIntBetween(0, 20)  
    point = Point(x, y) // Not running an IO, '=' instead of '<-'  
    _ <- Console.println(s"point=$point")  
  } yield ()
```

# Anatomy of `for` Comprehension

**for comprehension is not a for loop.**

It can be a for loop...

But it can handle **many other things**

like IO and... Seq, Option, Range...



# for Comprehension Types

```
val printRandomPoint: IO[Point] = {
  for {
    x      /* Int */ <- Random.nextIntBetween(0, 10)      /* IO[Int] */
    _      /* Unit */ <- Console.println(s"x=$x")          /* IO[Unit] */
    y      /* Int */ <- Random.nextIntBetween(0, 10)      /* IO[Int] */
    _      /* Unit */ <- Console.println(s"y=$y")          /* IO[Unit] */
    point  /* Point */ = Point(x, y)                      /* Point */
    _      /* Unit */ <- Console.println(s"point.x=${point.x}") /* IO[Unit] */
    _      /* Unit */ <- Console.println(s"point.y=${point.y}") /* IO[Unit] */
  } yield point /* Point */
} /* IO[Point] */
```

# for Comprehension **Type Rules**

	val <b>type</b>	operator	expression <b>type</b>
generation	A	<-	IO[A]
assignment	B	=	B

	for <b>comprehension type</b>	yield <b>expression type</b>
production	IO[R]	R

- Combines **only** IO[E, T], **no mix** with Seq[T], Option[T]...
- But it could be **only** Seq[T], **only** Option[T]...

# for Comprehension **Scopes**

```
val printRandomPoint: IO[Point] = {  
  for {  
    x <- Random.nextIntBetween(0, 10)           /* x */  
    _ <- Console.println(s"x=$x")               /* 0 */  
    y <- Random.nextIntBetween(0, 10)           /* | y */  
    _ <- Console.println(s"y=$y")               /* | 0 */  
    point = Point(x, y)                         /* 0 0 point */  
    _ <- Console.println(s"point.x=${point.x}") /* | | 0 */  
    _ <- Console.println(s"point.y=${point.y}") /* | | 0 */  
  } yield point                                /* | | 0 */  
}
```

# for Comprehension **Implicit Nesting**

```
val printRandomPoint: IO[Point] = {  
  for {  
    x <- Random.nextIntBetween(0, 10)  
    /* | */ _ <- Console.println(s"x=$x")  
    /* |   | */ y <- Random.nextIntBetween(0, 10)  
    /* |   |   | */ _ <- Console.println(s"y=$y")  
    /* |   |   |   | */ point = Point(x, y)  
    /* |   |   |   |   | */ _ <- Console.println(s"point.x=${point.x}")  
    /* |   |   |   |   |   | */ _ <- Console.println(s"point.y=${point.y}")  
  } /* |   |   |   |   |   |   | */ yield point  
}
```

# Writing a Small Application

# Saying Hello

```
val helloApp: IO[Unit] =  
  for {  
    _ <- Console.println("What's your name?")  
    name <- Console.readLine  
    _ <- Console.println(s"Hello $name!")  
  } yield ()
```

# Counting Down

```
val countdownApp: IO[Unit] =  
  for {  
    n <- readIntBetween(10, 100000)  
    _ <- countdown(n)  
  } yield ()  
  
def countdown(n: Int): IO[Unit] =  
  if n == 0 then  
    Console.println("BOOM!!!")  
  else  
    for {  
      _ <- Console.println(n)  
      _ <- /* RECURSE */ countdown(n - 1)  
    } yield ()
```

# Displaying Menu and Getting Choice

```
val displayMenu: IO[Unit] =  
  for {  
    _ <- Console.println("1) Hello")  
    _ <- Console.println("2) Countdown")  
    _ <- Console.println("3) Exit")  
  } yield ()  
  
val readChoice: IO[Int] = readIntBetween(1, 3)
```



# Launching Menu Item

```
def launchMenuItem(choice: Int): IO[Boolean] =  
  choice match {  
    case 1 => helloApp.map(_ => false)  
    case 2 => countDownApp.map(_ => false)  
    case 3 => IO.succeed(true)  
  }
```

# Looping over Menu

```
def mainApp: IO[Unit] =  
  for {  
    _ <- displayMenu  
    choice <- readChoice  
    exit <- launchMenuItem(choice)  
    _ <- if exit then IO.succeed(()) else /* RECURSE */ mainApp  
  } yield ()
```

# Reading Integer from Console

```
def readIntBetween(min: Int, max: Int): IO[Int] =  
  for {  
    _ <- Console.println(s"Enter a number between $min and $max")  
    i <- readInt  
    n <- if min <= i && i <= max then IO.succeed(i) else /* RECURSE */ readIntBetween(min, max)  
  } yield n  
  
def readInt: IO[Int] = Console.readLine.map(_.toInt)
```

# Just a Fancy Toy

- What's **good**
  - Easy to reason about with type safety 👍
  - Unlimited safe refactorings 👍
- What's **not so good**
  - Stack unsafe 💣
  - Do not handle exceptions, need a better error model 👎
  - Not testable 👎
  - Difficult to debug 👎

# Toward a Stack-Free Implementation

# Describing Operations with an ADT

```
trait IO[+A] { // `trait`, an interface on steroids
  // ...
}

object IO { // ...
  // `Op` enum has a type parameter A
  // An ADT with a type parameter is called a Generalized ADT (or GADT)
  enum Op[+A] extends IO[A] {
    case Succeed(result: () => A) extends Op[A]
    case FlatMap[A0, A](io: IO[A0], cont: A0 => IO[A]) extends Op[A]
  }
}
```

# Implementing Same Methods as Before

```
trait IO[+A] {  
  def flatMap[B](cont: A => IO[B]): IO[B] =  
    Op.FlatMap(this, cont)  
  
  def map[B](trans: A => B): IO[B] =  
    this.flatMap(a => IO.succeed(trans(a)))  
}  
  
object IO {  
  def succeed[A](a: => A): IO[A] = Op.Succeed(() => a)  
}
```

# Interpreting with Better Stack Safety

```
object Runtime {  
  def unsafeRun[A](io: IO[A]): A = {  
    io match {  
      case Op.Succeed(result /* () => A */) => result()  
  
      case Op.FlatMap(ioA0 /* IO[A0] */, cont /* A0 => IO[A] */) =>  
        val a0 = /* RECURSE */ unsafeRun(ioA0)  
        val ioA = cont(a0)  
        val a = /* TAIL_CALL_RECURSE */ unsafeRun(ioA)  
        a  
    }  
  }  
}
```



# Harder, Better, Faster, Stronger

— *Daft Punk*

# What About Real Life Applications?

- What we could possibly dream of for **real life applications**
  - Support for **asynchronicity**, **concurrency** and **interruptibility**
  - Consistent **error model** (expected vs. unexpected)
  - **Resiliency** and **resource safety**
  - Full **testability** with dependency injection
  - Easy **debugging**
  - **Performance** and **stack safety**
  - And still fully functional with **100% safe refactorings**
- *ZIO*, an easy to use Scala library, gives it to us!