Practical Functional Programming

Pure I/O

in Scala

Previously, **immutability**

Immutability

- Immutable **objects** (`case class`)
- Immutable **collections** (`Seq`, `IndexedSeq`, `Map`, `Set`)
- Immutable **options** (`Option`)
- Immutable **enumerations** (`enum`) aka Algebraic Data Types (**ADT**)
- Expressions
- Pattern Matching

Previously, **bank operations**

```scala
enum Operation {
  case Credit(account: Int, amount: Double)
  case Debit(account: Int, amount: Double)
  case Transfer(sourceAccount: Int, destinationAccount: Int, amount: Double)
}

case class Bank(accounts: Map[Int, Double]) {
  def process(operation: Operation): Bank = {
    operation match {
      case Credit(account, amount) => ???
      case Debit(account, amount) => ???
      case Transfer(sourceAccount, destinationAccount, amount) => ???
    }
  }
}
```

Modeling Bank Operations

```scala
// Immutable class (`case class`) using an immutable collection (`Map`)
case class Bank(name: String, accounts: Map[Int, Double]) {
  // ...
}

// ADT (Algebraic Data Type) or just an `enum` on steroids
enum Operation {
  case Credit(account: Int, amount: Double)
  case Debit(account: Int, amount: Double)
  case Transfer(sourceAccount: Int, destinationAccount: Int, amount: Double)
}
```

Processing Bank Operation

```
case class Bank(name: String, accounts: Map[Int, Double]) {
  def process(operation: Operation): Bank = {
    operation match { // Pattern matching (`match`)
      case Credit(account, amount) => // Extract into `account` and `amount`
        val updatedAccounts = this.accounts.updatedWith(account, _ + amount)
        // Immutable variable (`val`)
        // `_ + amount` is equivalent to `a => a + amount`
        this.copy(accounts = updatedAccounts)
        // `updatedAccounts` passed as argument to the `account` parameter

      // Other cases are similar
    }
  }
}
```

Processing Operations Sequentially

```scala
val bank = Bank("My Bank", Map(1 -> 100.0, 2 -> 200.0))
// No need for `new` keyword

val finalBank = bank
  .process(Credit(account = 1, amount = 30.0))
  .process(Debit(account = 2, amount = 10.0))
  .process(Transfer(sourceAccount = 1, destinationAccount = 2, amount = 10.0))
```

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

```scala
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.printLine("What's player 1 name?")
    val player1 = Console.readLine()
    Console.printLine("What's player 2 name?")
    val player2 = Console.readLine();
    Console.printLine(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

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

```scala
object ConsoleApp {
  def main(args: Array[String]): Unit = {
    Console.printLine("What's player 1 name?")
    Console.printLine("What's player 2 name?")
    val player2 = Console.readLine();
    Console.printLine(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

```scala
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

```scala
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.printLine("What's your name?").flatMap { _ =>
      Console.readLine.flatMap { name =>
        Console.printLine(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

```scala
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` 😈

```scala
val welcomeNewPlayer: IO[Unit] =
  Console.printLine("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.printLine(s"Welcome $name, you start at coordinates ($x, $y, $z).")
          }
        }
      }
    }
  }
```

Flatten Them All 😇

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

Syntactic sugar that calls `flatMap` and `map`

Intermediary Variable

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

`for` Comprehension **Type Rules**

	`val` **type**	**operator**	**expression type**
generation	`A`	`<-`	`IO[A]`
assignment	`B`	`=`	`B`

	`for` **comprehension type**	`yield` **expression type**
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.printLine(s"x=$x")                    /*  O              */
    y <- Random.nextIntBetween(0, 10)                  /*  |    y         */
    _ <- Console.printLine(s"y=$y")                    /*  |    O         */
    point = Point(x, y)                                /*  O    O    point */
    _ <- Console.printLine(s"point.x=${point.x}")      /*  |    |    O    */
    _ <- Console.printLine(s"point.y=${point.y}")      /*  |    |    O    */
  } yield point                                        /*  |    |    O    */
}
```

for Comprehension **Implicit Nesting**

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

Writing a Small Application

Saying Hello

```
val helloApp: IO[Unit] =
  for {
    _ <- Console.printLine("What's your name?")
    name <- Console.readLine
    _ <- Console.printLine(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.printLine("BOOM!!!")
  else
    for {
      _ <- Console.printLine(n)
      _ <- /* RECURSE */ countdown(n - 1)
    } yield ()
```

Displaying Menu and Getting Choice

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

val readChoice: IO[Int] = readIntBetween(1, 3)
```

Launching Menu Item

```scala
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

```scala
def readIntBetween(min: Int, max: Int): IO[Int] =
  for {
    _ <- Console.printLine(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-Freer 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!