Immutable

Pure I/O

in Java... and in Scala with ZIO

Previously on Practical Immutability...

- Immutable Classes
- Immutable Collections and Options
- Immutable variables
- Expressions
- Algebraic Data Types (ADT)
- Pattern Matching

Functional Programming

- FP is programming with **functions**.
- Functions are:
  - **Deterministic**: same arguments implies same result
  - **Total**: result always available for arguments
  - **Pure**: no side-effects, only effect is computing result
- A consequence of FP is **referential transparency**.

Referential Transparency

- FP programs are **referentially transparent**.
  - **Typical refactorings cannot break a working program** 😄.
- Applies to the following refactorings:
  - 🔧 **Extract Variable**
  - 🔧 **Inline Variable**
  - 🔧 **Extract Method**
  - 🔧 **Inline Method**

Refactorings Break Impure 😈 Programs

Console Operations

```
public class Console {
    public static String getStrLn() {
        return new Scanner(System.in).nextLine();
    }

    public static void putStrLn(final String line) {
        System.out.println(line);
    }
}
```

A Working Program

```
public class ConsoleApp {
    public static void main(String[] args) {
        putStrLn("What's player 1 name?");
        final String player1 = getStrLn();
        putStrLn("What's player 2 name?");
        final String player2 = getStrLn();
        putStrLn(String.format("Players are %s and %s.", player1, player2));
    }
}
```

```
What's player 1 name?
> Paul
What's player 2 name?
> Mary
Players are Paul and Mary.
```

Broken Extract Variable Refactoring

```
public class BrokenExtractVariableConsoleApp {
    public static void main(String[] args) {
        final String s = getStrLn();
        putStrLn("What's player 1 name?");
        final String player1 = s;
        putStrLn("What's player 2 name?");
        final String player2 = s;
        putStrLn(String.format("Players are %s and %s.", player1, player2));
    }
}
```

```
> Paul
What's player 1 name?
What's player 2 name?
Players are Paul and Paul.
```

Broken Inline Variable Refactoring

```
public class BrokenInlineVariableConsoleApp {
    public static void main(String[] args) {
        putStrLn("What's player 1 name?");
        putStrLn("What's player 2 name?");
        final String player2 = getStrLn();
        putStrLn(String.format("Players are %s and %s.", getStrLn(), 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

in Java with *Immutables* and *Vavr*

Describing a Program

```
public abstract class Program<A> { /* ... */ }
```

- Describes a **program** performing I/Os
- When run, will eventually yield a **result** of type `A`

Program As Immutable Object

```
@Value.Immutable
public abstract class Program<A> { // ...
    @Value.Parameter
    public abstract Supplier<A> unsafeAction();

    public static <A> Program<A> of(final Supplier<A> unsafeAction) {
        return ImmutableProgram.of(unsafeAction);
    } // ...
}
```

Program Yielding a Value

```
@Value.Immutable
public abstract class Program<A> { // ...
    public static <A> Program<A> yield(final A a) {
        return Program.of(() -> a);
    } // ...
}
```

Chaining Programs

```
@Value.Immutable
public abstract class Program<A> { // ...
    public <B> Program<B> thenChain(final Function<A, Program<B>> f) {
        final Program<A> pa = this;

        return Program.of(() -> {
            final A a = pa.unsafeAction().get();
            final Program<B> pb = f.apply(a);
            final B b = pb.unsafeAction().get();
            return b;
        });
    } // ...
}
```

Transforming Result of Program

```
@Value.Immutable
public abstract class Program<A> { // ...
    public <B> Program<B> thenTransform(final Function<A, B> f) {
        final Program<A> pa = this;

        return pa.thenChain(a -> {
            final B b = f.apply(a);
            final Program<B> pb = Program.yield(b);
            return pb;
        });
    } // ...
}
```

Elementary Programs

```
public class Console {
    public static Program<String> getStrLn() {
        return Program.of(() -> {
            final String line = new Scanner(System.in).nextLine();
            return line;
        });
    }

    public static Program<Unit> putStrLn(final String line) {
        return Program.of(() -> {
            System.out.println(line);
            return Unit.of();
        });
    }
}
```

A Value Containing Void (`Unit`)

```
@Value.Immutable(singleton = true)
public abstract class Unit {
    public static Unit of() {
        return ImmutableUnit.of();
    }
}
```

- Cannot use `Void`
- Cannot create instances (`private` constructor 😟)
- Can just use `null` 😈

Instantiating a Program

```
public class ConsoleApp {
    public static final Program<Unit> helloApp =
            putStrLn("What's you name?").thenChain(__ -> {
                return getStrLn().thenChain(name -> {
                    return putStrLn("Hello " + name + "!");
                });
            });

    public static void main(String[] args) {
        final Program<Unit> program = helloApp;
    }
}
```

But Program Does Not Run

```
public class ConsoleApp {
    // ...
    public static void main(String[] args) {
        final Program<Unit> program = helloApp;
        System.out.println(program);
    }
}
```

- Will print something like `Program{unsafeAction=pureio.console.pure.Program$$Lambda$3/511754216@5197848c}`
- This is just an **immutable object**, it does no side-effect, it's **pure** 😇.
- Need an **interpreter** to run!

Interpreting a Program

```
@Value.Immutable
public abstract class Program<A> {
    // ...

    public static <A> A unsafeRun(final Program<A> program) {
        return program.unsafeAction().get();
    }
}
```

Running a Program

```
public class ConsoleApp {
    // PURE ...
    public static void main(String[] args) {
        final Program<Unit> program = helloApp; // PURE
        Program.unsafeRun(program); // IMPURE!!! But that's OK!
    }
}
```

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

Counting Down

```
public static final Program<Unit> countdownApp =
        getIntBetween(10, 100000).thenChain(n -> {
            return countdown(n);
        });

public static Program<Unit> countdown(final int n) {
    if (n == 0) {
        return putStrLn("BOOM!!!");
    } else {
        return putStrLn(Integer.toString(n)).thenChain(__ -> {
            return /* RECURSE */ countdown(n - 1);
        });
    }
}
```

Diplaying Menu and Getting Choice

```
public static final Program<Unit> displayMenu =
        putStrLn("Menu")
                .thenChain(__ -> putStrLn("1) Hello"))
                .thenChain(__ -> putStrLn("2) Countdown"))
                .thenChain(__ -> putStrLn("3) Exit"));

public static final Program<Integer> getChoice =
        getIntBetween(1, 3);
```

Launching Menu Item

```
public static Program<Boolean> launchMenuItem(final int choice) {
    switch (choice) {
        case 1: return helloApp.thenTransform(__ -> false);
        case 2: return countdownApp.thenTransform(__ -> false);
        case 3: return Program.yield(true); // Should exit
        default: throw new IllegalArgumentException("Unexpected choice");
    }
}
```

Looping over Menu

```
public static Program<Unit> mainApp() {
    return displayMenu.thenChain(__ -> {
        return getChoice.thenChain(choice -> {
            return launchMenuItem(choice).thenChain(exit -> {
                if (exit) {
                    return Program.yield(Unit.of());
                } else {
                    return /* RECURSE */ mainApp();
                }
            });
        });
    });
}
```

Parsing an Integer with a Total Function

```
public static Option<Integer> parseInt(final String s) {
    return Try.of(() -> Integer.valueOf(s)).toOption();
}
```

- `parseInt` is defined for any `String`, it's **total**.
- No exception! 😉

Expression	Result
`parseInt("3")`	`Some(3)`
`parseInt("a")`	`None`

Getting Integer from Console

```
public static Program<Integer> getInt() {
    return getStrLn()
            .thenTransform(s -> parseInt(s))
            .thenChain(maybeInt -> {
                return maybeInt.isDefined() ? Program.yield(maybeInt.get()) : /* RECURSE */ getInt();
            });
}

public static Program<Integer> getIntBetween(final int min, final int max) {
    final String message = String.format("Enter a number between %d and %d", min, max);
    return putStrLn(message).thenChain(__ -> {
        return getInt().thenChain(i -> {
            return min <= i && i <= max ? Program.yield(i) : /* RECURSE */ getIntBetween(min, max);
        });
    });
}
```

Just a Toy

- What's **good** 👍
  - Rather efficient
  - Unlimited refactorings
- What's **not so good** 👎
  - Not stack safe
  - Nesting can be annoying (extract variables and methods!)
  - Not testable

Business-Ready Pure IO

in Scala with *ZIO*

ZIO

Type-safe, composable, asynchronous and concurrent programming for Scala

— *https://scalaz.github.io/scalaz-zio/*

- **Synchronicity** and **Asynchronicity** (handled reactively), combine seamlessly
- **Concurrency**, based on lightweight fibers
- **Resiliency**, can recover from errors
- **Interruptibility**, can interrupt any program
- **Resource Safety**, ensure resources will never leak (threads, sockets, file handles...)
- **Performance**, extremely fast given features and strong guarantees
- And also **Composability** and **Testability**

IO[E, A]

```
IO[E, A] // IO<E, A>   E = Error, A = Result
```

- An immutable object that **describes** a **program performing side-effects**.
- An `IO` does nothing, it's just a **value** holding a program.
- It must be interpreted by a **runtime system** or **RTS**
- Only when **run** by the RTS, it will either
  - fail with an **error** of type `E`,
  - or eventually produce a **result** of type `A`.

Hello World!

```
object HelloWorldApp extends App {
  // Wraps synchronous (blocking) side-effecting code in an IO
  val helloWorld: IO[Nothing, Unit] =
    IO.effectTotal(/* () => */ Console.println("Hello World!"))
    // The IO just holds a lambda but does not run it for now.

  def run(args: List[String]): IO[Nothing, Int] = {
    helloWorld.either.fold(_ => 1, _ => 0)
  }
}
```

Wrapping in `IO`

Success in IO

```
val success: IO[Nothing, Int] = IO.succeed(42)
val successLazy: IO[Nothing, Int] = IO.succeedLazy(/* () => */ 40 + 2)

// Will never fail (Nothing)
// Will always succeed with result 42 (Int)
```

Failure in IO

```
val failure: IO[String, Nothing] = IO.fail("Failure")
// Will always fail with error "Failed" (String)
// will never succeed (Nothing)

val exceptionFailure: IO[IllegalStateException, Nothing] =
  IO.fail(new IllegalStateException("Failure"))
// Error can be an exception (but just as a value, never thrown!)
```

Synchronous in `IO`

```
def randomBetween(min: Int, max: Int): IO[Nothing, Int] = {
  // Side-effecting code updates the state of a random generator,
  // and returns a random number (Int).
  // It can never fail (Nothing).
  IO.effectTotal(/* () => */ Random.nextInt(max - min) + min)
}

def putStrLn(line: String): IO[Nothing, Unit] = {
  // Side-effecting code prints a line,
  // and returns void (Unit).
  // It can never fail (Nothing).
  IO.effectTotal(/* () => */ scala.Console.println(line))
}
```

Synchronous, Exception-Throwing in `IO`

```
def getStrLn: IO[IOException, String] = {
  // Side-effecting code reads from keyboard until a line is available,
  // and returns the line (String).
  // It might throw an IOException. IO catches exception,
  // and translates it into a failure containing the error (IOException).
  // IOException is neutralized, it is NOT propagated but just used as a value.
  IO.effect(/* () => */ scala.io.StdIn.readLine()).refineOrDie {
    case e: IOException => e
  }
}
```

Asynchronous in IO

```
object Calculator {
  private lazy val executor = Executors.newScheduledThreadPool(5)

  def add(a: Int, b: Int): IO[Nothing, Int] = {
    IO.effectAsync { (callback: IO[Nothing, Int] => Unit) =>
      val completion: Runnable = { () => callback(IO.succeedLazy(a + b)) }
      executor.schedule(completion, 5, TimeUnit.SECONDS)
    }
  }
}
```

Asynchronous, Interruptible in `IO`

```
object Calculator {
  private lazy val executor = Executors.newScheduledThreadPool(5)

  def add(a: Int, b: Int): IO[Nothing, Int] = {
    IO.effectAsyncInterrupt { (callback: IO[Nothing, Int] => Unit) =>
      val complete: Runnable = { () => callback(IO.succeedLazy(a + b)) }
      val eventualResult = executor.schedule(complete, 5, TimeUnit.SECONDS)
      val canceler = IO.effectTotal(eventualResult.cancel(false))
      Left(canceler)
    }
  }
}
```

Combining `IO`s

Transforming IO (map)

```
val randomLetter: IO[Nothing, Char] =
  randomBetween('A', 'Z').map { i /* Int */ =>
    i.toChar /* Char */
  }
```

Chaining `IO`s (broken `map`)

```
val printRolledDiceWRONG: IO[Nothing, IO[Nothing, Unit]] =
  randomBetween(1, 6).map { dice /* Int */ =>
    putStrLn(s"Dice shows $dice") /* IO[Nothing, Unit] */
  }
```

- Wrong **nested** type `IO[Nothing, IO[Nothing, Unit]]`
- Needs to be made **flat** somehow as `IO[Nothing, Unit]`

Chaining IOs (flatMap)

```
val printRolledDice: IO[Nothing, Unit] =
  randomBetween(1, 6).flatMap { dice /* Int */ =>
    putStrLn(s"Dice shows $dice") /* IO[Nothing, Unit] */
  }
```

Chaining and Transforming IOs

```
randomBetween(0, 20).flatMap { x =>
  randomBetween(0, 20).map { y =>
    Point(x, y)
  }
}
```

Pyramid of `maps` and `flatMaps` 😈

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

Flatten Them All 😇

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

Intermediary Variable

```
val printRandomPoint: IO[Nothing, Unit] =
  for {
    x <- randomBetween(0, 20)
    y <- randomBetween(0, 20)
    point = Point(x, y) // Not running an IO, '=' instead of '<-'
    _ <- putStrLn(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`, `Future`...

`for` Comprehension **Types**

```
val printRandomPoint: IO[Nothing, Point] = {
  for {
    x     /* Int   */ <- randomBetween(0, 10)            /* IO[Nothing, Int]  */
    _     /* Unit  */ <- putStrLn(s"x=$x")               /* IO[Nothing, Unit] */
    y     /* Int   */ <- randomBetween(0, 10)            /* IO[Nothing, Int]  */
    _     /* Unit  */ <- putStrLn(s"y=$y")               /* IO[Nothing, Unit] */
    point /* Point */ =  Point(x, y)                     /* Point             */
    _     /* Unit  */ <- putStrLn(s"point.x=${point.x}") /* IO[Nothing, Unit] */
    _     /* Unit  */ <- putStrLn(s"point.y=${point.y}") /* IO[Nothing, Unit] */
  } yield point /* Point */
} /* IO[Nothing, Point] */
```

`for` Comprehension **Type Rules**

	`val` **type**	**operator**	**expression type**
generator	`A`	`<-`	`IO[E, A]`
assignment	`B`	`=`	`B`

	`for` **comprehension type**	`yield` **expression type**
production	`IO[E, R]`	`R`

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

`for` Comprehension **Scopes**

```
val printRandomPoint: IO[Nothing, Point] = {
  for {
    x <- randomBetween(0, 10)                 /*  x                  */
    _ <- putStrLn(s"x=$x")                    /*  0                  */
    y <- randomBetween(0, 10)                 /*  |    y             */
    _ <- putStrLn(s"y=$y")                    /*  |    0             */
    point = Point(x, y)                       /*  0    0    point   */
    _ <- putStrLn(s"point.x=${point.x}")      /*  |    |    0       */
    _ <- putStrLn(s"point.y=${point.y}")      /*  |    |    0       */
  } yield point                               /*  |    |    0       */
}
```

`for` Comprehension **Implicit Nesting**

```
val printRandomPoint: IO[Nothing, Point] = {
  for {
       x <- randomBetween(0, 10)
    /* | */ _ <- putStrLn(s"x=$x")
    /* |    | */ y <- randomBetween(0, 10)
    /* |    |    | */ _ <- putStrLn(s"y=$y")
    /* |    |    |    | */ point = Point(x, y)
    /* |    |    |    |    | */ _ <- putStrLn(s"point.x=${point.x}")
    /* |    |    |    |    |    | */ _ <- putStrLn(s"point.y=${point.y}")
  } /* |    |    |    |    |    |    | */ yield point
}
```

Conditions and Loops

Behaving Conditionally

```
def describeNumber(n: Int): IO[Nothing, Unit] = {
  for {
    _ <- if (n % 2 == 0) putStrLn("Even") else putStrLn("Odd")
    _ <- if (n == 42) putStrLn("The Anwser") else IO.unit
  } yield ()
}
```

Looping with **Recursion** 😨

```
def findName(id: Int): IO[Nothing, String] =
  IO.succeedLazy(s"Name $id")

def findNames(ids: List[Int]): IO[Nothing, List[String]] = {
  ids match {
    case Nil => IO.succeed(Nil)

    case id :: restIds =>
      for {
        name      /* String       */ <- findName(id)       /* IO[Nothing, String]       */
        restNames /* List[String] */ <- findNames(restIds) /* IO[Nothing, List[String]] */
      } yield name :: restNames /* List[String] */
  }
}
```

Looping with `foreach`

```
def findName(id: Int): IO[Nothing, String] =
  IO.succeedLazy(s"Name $id")

def findNames(ids: List[Int]): IO[Nothing, List[String]] =
  IO.foreach(ids) { id => findName(id) }
```

- Recursion can be hard to read
- Prefer using simpler alternatives whenever possible
  - `IO.foreach`, `IO.collectAll`, `IO.reduceAllPar`, `IO.mergeAll`
  - Or `IO.foreachPar`, `IO.collectAllPar`, `IO.reduceAllPar`, `IO.mergeAllPar` in **parallel** 👍

Further with *ZIO*

Keeping Resource Safe

```
class Resource {
  def close: IO[Nothing, Unit] = ???
  def read: IO[Int, String] = ???
}

object Resource {
  def open(name: String): IO[Int, Resource] = ???
}

val program: IO[Nothing, Unit] =
  IO.bracket(Resource.open("hello"))(_.close) { resource =>
    for {
      line <- resource.read
      _ <- putStrLn(line)
    } yield ()
  }
```

Retrying After Error

```
object NameService {
  def find(id: Int): IO[Int, String] = ???
}

val retrySchedule = Schedule.recurs(3) && Schedule.exponential(1.second)

val program =
  for {
    name <- NameService.find(1).retry(retrySchedule)
    _ <- putStrLn(s"name=$name")
  } yield ()
```

- `retry` repeats in case of **failure**.
- There also exists `repeat` that repeats in case of **success**.

Forking and Interrupting

```
val analyze: IO[Nothing, String] = ???
val validate: IO[Nothing, Boolean] = ???

val program: IO[Nothing, String] =
  for {
    analyzeFiber <- analyze.fork
    validateFiber <- validate.fork
    validated <- validateFiber.join
    _ <- if (validated) IO.unit else analyzeFiber.interrupt
    analysis <- analyzeFiber.join
    _ <- putStrLn(analysis)
  } yield analysis
```

But There's Much More in *ZIO*

- **Streaming**
  - `Stream`, a lazy, concurrent, asynchronous source of values
  - `Sink`, a consumer of values from a `Stream`, which may produces a value when it has consumed enough
- **Software Transactional Memory** (STM)
- **Low Level Concurrency**
  - `Promise`, a variable that may be set a single time, and awaited on by many fibers
  - `Queue`, an asynchronous queue that never blocks
  - `FiberLocal`, a variable whose value depends on the fiber that accesses it
  - `Semaphore`, a semaphore
  - `Ref`, a mutable reference to a value