Practical

Pure I/O

in Java with help of *Immutables* and *Vavr*

Previously on Practical Immutability...

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

What Functional Programming Is About

- Functional Programming (FP) is programming with **functions**.
  - **Deterministic**: same arguments implies same result
  - **Total**: result always available for arguments, no exception
  - **Pure**: no side-effects, only effect is computing result
- A benefit of FP is **referential transparency**.

What Referential Transparency Brings

- **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

```java
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

```java
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

```java
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

```java
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

Describing a Program

```java
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

```java
@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

```java
@Value.Immutable
public abstract class Program<A> { // ...
    public <B> Program<B> thenChain(final Function<A, Program<B>> f) {
        final Program<A> pa = this;
        final Program<B> pb = Program.of(() -> {
            final A a = pa.unsafeAction().get();
            final Program<B> _pb = f.apply(a);
            final B b = _pb.unsafeAction().get();
            return b;
        });
        return pb;
    } // ...
}
```

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

Elementary `Console` 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 😲

```java
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

```java
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);
        });
    }
}
```

Displaying 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 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

```
public interface Program<A> { // ...
    @Value.Immutable abstract class Of<A> implements Program<A> {
        @Value.Parameter abstract Supplier<A> unsafeRun(); // ...
    }
    @Value.Immutable abstract class Yield<A> implements Program<A> {
        @Value.Parameter abstract A value(); // ...
    }
    @Value.Immutable abstract class ThenChain<A, B> implements Program<B> {
        @Value.Parameter abstract Program<A> pa();
        @Value.Parameter abstract Function<A, Program<B>> f(); // ...
    } // ...
}
```

Implementing Same Methods as Before

```java
public interface Program<A> { // ...
    static <A> Program<A> of(final Supplier<A> unsafeRun) {
        return Of.of(unsafeRun);
    }
    static <A> Program<A> yield(final A a) {
        return Yield.of(a);
    }
    default <B> Program<B> thenChain(final Function<A, Program<B>> f) {
        return ThenChain.of(this, f);
    }
    default <B> Program<B> thenTransform(final Function<A, B> f) {
        return ThenChain.of(this, a -> Yield.of(f.apply(a)));
    } // ...
}
```

Interpreting with Better Stack Safety

```
static <A> A unsafeRun(final Program<A> program) {
    Program<A> current = program;
    do { // Run all steps (mostly) stack-free even for recursion (trampoline)
        if (current instanceof Of) {
            final Of<A> of = (Of<A>) current;
            return of.unsafeRun().get();                           // RETURN result
        } else if (current instanceof Yield) {
            final Yield<A> yield = (Yield<A>) current;
            return yield.value();                                  // RETURN result
        } else if (current instanceof ThenChain) {
            final ThenChain<Object, A> thenChain = (ThenChain<Object, A>) current;
            final Object a = /* RECURSE */ unsafeRun(thenChain.pa()); // EXECUTE current step
            current = thenChain.f().apply(a);                      // GET remaining steps (continuation)
        } else {
            throw new IllegalArgumentException("Unexpected Program");
        }
    } while (true);
}
```

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!