More ... Practical

Immutability

in Java with *Immutables* and *Vavr*

Previously on Practical Immutability...

- **Immutable Classes** with *Immutables*
  - Creating and modifying create a new instance
  - Comparing by value
  - Preventing `null` attributes
  - Ensuring consistency with class invariant
- **Immutable Collections and Options** with *Vavr*
  - `Seq`, `IndexedSeq`, `Set`, `Map`, `Option`…
  - `map`, `filter`, `forAll`, `removeFirst`, `indexWhere`, `update`, `count`…

But there is more to immutability than objects, collections and options

Immutability of Variables

- Mutability of **variables** `!=` Mutability of **objects**
- Immutability of **objects**
  - Cannot mutate the fields of the object or collection
  - As seen so far
- Immutability of **variables** (local variable, parameter)
  - Cannot change the value (or reference) contained in the variable
  - `final` vs. ~~`final`~~

Mutability Combinations

	Immutable Object	**Mutable Object**
`final` **Variable**	😄	😐 if stricly local 😨 otherwise
non-`final` **Variable**	😐 stricly local	👿

Expressions

in Java ... and with *Vavr*

Expressions vs. Instructions

- An **expression** evaluates to a value
  - Value can be directly assigned to a `final` variable
  - Expressions, when *pure* 😇, do not cause any side-effect
- An **instruction** does something and has no value
  - Instructions always cause side-effects

`final`, `final` Everywhere

- As many `final` as possible to **reduce moving parts**
- Somewhat controversial for other than local variables

Type of variable	**Benefit of `final`**
Local variable	Emulates **expressions** 👍 Prevents confusing reassignment
Parameter	Prevents rare reassignment
`for` enhanced loop variable	Prevents rare reassignment
`catch` clause variable	Prevents rare reassignment

… ? … : … Expression

```
final String status = enabled ? "On" : "Off";
```

- An actual conditional expression!
- Only one of that kind in Java
- Only for very simple one-liners

Emulating `if` Expression

```java
final String mood; // No default value

// Every branch either assigns value or fails
// Compiler is happy
if (1 <= mark && mark <= 3) {
    mood = "Bad";
} else if (mark == 4) {
    mood = "OK";
} else if (5 <= mark && mark <= 7) {
    mood ="Good";
} else {
    throw new AssertionError("Unexpected mark (" + mark + ")");
}
```

Emulating `switch` Expression

```java
final int mark;

switch (color) {
    case RED: mark = 1; break;
    case YELLOW: mark = 3; break;
    case GREEN: mark = 5; break;

    default:
        throw new AssertionError("Unexpected color (" + color + ")");
}
```

Another `Try` Expression with *Vavr*

```
final Try<Integer> triedNumber = Try.of(() -> Integer.parseInt(input))
        .filter(i -> i > 0)
        .map(i -> i * 10);
```

`input`	**`triedNumber` prints as**
`"3"`	`Success(30)`
`"-10"`	`Failure(java.util.NoSuchElementException: Predicate does not hold for -10)`
`"WRONG"`	`Failure(java.lang.NumberFormatException: For input string: "WRONG")`

Try to Option

```
final Try<Integer> triedNumber = Try.of(() -> Integer.parseInt(input))
        .filter(i -> i > 0)
        .map(i -> i * 10);

final Integer defaultedNumber = triedNumber.getOrElse(0);
final Option<Integer> maybeNumber = triedNumber.toOption();
```

input	defaultedNumber **prints as**	maybeNumber **prints as**
"3"	30	Some(30)
"-10"	0	None
"WRONG"	0	None

Algebraic Data Types

with *Immutables*

What is that 💩?

Algebraic Data Type

- **ADT** in short
- Also called **discriminated union** in some other world
- Somehow, **`enum`** **on steroids**
    - Some alternatives might hold one or more **attributes**
    - Attributes may vary in number and in type from one alternative to another

`Direction` enumeration

```
public enum Direction {
    Up,
    Down,
    Left,
    Right
}
```

Position class

```
@Value.Immutable
public abstract class Position {
    @Value.Parameter
    public abstract int x();

    @Value.Parameter
    public abstract int y();

    public static Position of(final int x, final  int y) {
        return ImmutablePosition.of(x, y);
    }
}
```

Updating Position with Direction

```
@Value.Immutable
public abstract class Position { // ...
    public Position move(final Direction direction) {
        switch (direction) {
            case Up: return ImmutablePosition.copyOf(this).withY(y() - 1);
            case Down: return ImmutablePosition.copyOf(this).withY(y() + 1);
            case Left: return ImmutablePosition.copyOf(this).withX(x() - 1);
            case Right: return ImmutablePosition.copyOf(this).withX(x() + 1);
            default: throw new IllegalArgumentException(
                        String.format("Unknown Direction (%s)", direction));
        }
    } // ...
}
```

Encoding `Action` ADT

```java
public interface Action {
    @Value.Immutable(singleton = true)
    abstract class Sleep implements Action {
        public static Sleep of() { return ImmutableSleep.of(); }
    }
    @Value.Immutable
    abstract class Walk implements Action {
        @Value.Parameter public abstract Direction direction();
        public static Walk of(final Direction direction) { return ImmutableWalk.of(direction); }
    }
    @Value.Immutable
    abstract class Jump implements Action {
        @Value.Parameter public abstract Position position();
        public static Jump of(final Position position) { return ImmutableJump.of(position); }
    }
}
```

Instantiating `Action` ADT

```
final Seq<Action> actions = List.of(
        Jump.of(Position.of(5, 8)),
        Walk.of(Up),
        Sleep.of(),
        Walk.of(Right)
);
```

Player class

```
@Value.Immutable
public abstract class Player {
    @Value.Parameter
    public abstract Position position();

    public static Player of(final Position position) {
        return ImmutablePlayer.of(position);
    }
}
```

Updating Player with Action

```
@Value.Immutable
public abstract class Player { // ...
    public Player act(final Action action) {
        if (action instanceof Sleep) {
            return this;
        } else if (action instanceof Walk) {
            final Walk walk = (Walk) action;
            return Player.of(position().move(walk.direction()));
        } else if (action instanceof Jump) {
            final Jump jump = (Jump) action;
            return Player.of(jump.position());
        } else {
            throw new IllegalArgumentException(String.format("Unknown Action (%s)", action));
        }
    } // ...
}
```

Applying Successive `Actions`

```
final Player initialPlayer = Player.of(Position.of(1, 1));

final Seq<Action> actions = List.of(
        Jump.of(Position.of(5, 8)), Walk.of(Up), Sleep.of(), Walk.of(Right));

final Player finalPLayer = actions.foldLeft(initialPlayer, Player::act);
final Seq<Player> players = actions.scanLeft(initialPlayer, Player::act);
```

- `finalPlayer` prints as: `Player{position=Position{x=6, y=7}}`
- `players` prints as: `List(Player{position=Position{x=1, y=1}}, Player{position=Position{x=5, y=8}}, Player{position=Position{x=5, y=7}}, Player{position=Position{x=5, y=7}}, Player{position=Position{x=6, y=7}})`

Visitor Pattern `ActionVisitor`

```
public interface ActionVisitor<T, R> {
    R visitSleep(Sleep sleep, T t);
    R visitWalk(Walk walk, T t);
    R visitJump(Jump jump, T t);
}
```

`Action` Made Visitable

```java
public interface Action {
    <R, T> R accept(ActionVisitor<T, R> visitor, T t); // ...
    abstract class Sleep implements Action { // ...
        public <R, T> R accept(final ActionVisitor<T, R> visitor, final T t) {
            return visitor.visitSleep(this, t);
        } // ...
    } // ...
    abstract class Walk implements Action { // ...
        public <R, T> R accept(final ActionVisitor<T, R> visitor, final T t) {
            return visitor.visitWalk(this, t);
        } // ...
    } // ...
    abstract class Jump implements Action { // ...
        public <R, T> R accept(final ActionVisitor<T, R> visitor, final T t) {
            return visitor.visitJump(this, t);
        } // ...
    }
}
```

Updating `Player` with `Action` using Visitor

```
@Value.Immutable
public abstract class Player { // ...
    private static final ActionVisitor<Player, Player> ACT_VISITOR = new ActionVisitor<Player, Player>() { // ...
        public Player visitSleep(final Sleep sleep, final Player player) {
            return player;
        } // ...
        public Player visitWalk(final Walk walk, final Player player) {
            return Player.of(player.position().move(walk.direction()));
        } // ...
        public Player visitJump(final Jump jump, final Player player) {
            return Player.of(jump.position());
        }
    };
    public Player act(final Action action) {
        return action.accept(ACT_VISITOR, this);
    } // ...
}
```

Pattern Matching

with *Vavr*

From `switch` to `Match` Expression

```java
import static io.vavr.API.*;
// ...
final String label = Match(number).of(
        Case($(0), "Zero"),
        Case($(1), "One"),
        Case($(2), "Two"),
        Case($(), "More")
);
```

`Match`, a `switch` on steroids

- `Match` is an **expression** compared to `switch`
- Many ways to **match a value**
- Might **extract one or more values**
- First match wins and gives the value of the expression
- Extracted values can be passed to a lambda expression and used to produce the value

`Case`, a `case` on steroids

Case form	What it matches and extracts
`$()`	Matches **anything** May extract the matching value
`$(1)`	Matches by **equality**
`$(i -> i > 0)`	Matches by **condition** May extract the matching value
`$Some($())`	Matches by **pattern** May extract matching values from pattern

Matching by Condition

```
import static io.vavr.Predicates.*;
// ...
final String label = Match(number).of(
        Case($(0), "Zero"),
        Case($(n -> n < 0), "Negative"),
        Case($(isIn(19, 23, 29)), "Chosen Prime"),
        Case($(i -> i % 2 == 0), i -> String.format("Even (%d)", i)),
        Case($(), i-> String.format("Odd (%d)", i))
);
```

Matching by Pattern

```
import static io.vavr.Patterns.*;
// ...
final String label = Match(maybeNumber).of(
        Case($Some($(0)), "Zero"),
        Case($Some($(i -> i < 0)), i -> String.format("Negative (%d)", i)),
        Case($Some($(i -> i > 0)), i -> String.format("Positive (%d)", i)),
        Case($None(), "Absent")
);
```

Could be on `Try` too, using `$Success` and `$Failure`

`Action` Custom Patterns

```
@Patterns
public interface Action {
    // ...
    @Unapply
    static Tuple0 Sleep(final Sleep sleep) {
        return Tuple.empty();
    }
    @Unapply
    static Tuple1<Position> Jump(final Jump jump) {
        return Tuple.of(jump.position());
    }
    @Unapply
    static Tuple1<Direction> Walk(final Walk walk) {
        return Tuple.of(walk.direction());
    }
}
```

Updating `Player` with `Action` using Pattern Matching

```
import static /*...*/ActionPatterns.*;
// ...
@Value.Immutable
public abstract class Player { // ...
    public Player act(final Action action) {
        return Match(action).of(
                Case($Sleep, () -> this),
                Case($Walk($()), direction -> ImmutablePlayer.of(position().move(direction))),
                Case($Jump($()), position -> ImmutablePlayer.of(position))
        );
    } // ...
}
```

To immutability... and beyond!

— Buzz Lightyear

More Types...

- `Either<E, R>` used traditionally to represent result and error alternative in a type
  - Either the right **result** of type `R` (`Right`, `$Right`)
  - or a left **error** of type `E` (`Left`, `$Left`)
- `Tuple0`, `Tuple1<A>`, `Tuple2<A, B>`, `Tuple3<A, B, C>` ...
  - Empty tuple (*unit*), singles, pairs, triples...

There is no Silver Bullet

- **Immutability pays off** even at small scale
  - Many no-brainers. If it's never mutated, make it immutable!
  - *Immutables* objects and *Vavr* collections are cool!
  - Code will be really more concise (more but simpler classes).
  - Concurrency and immutability is a match made in heaven!
- **Do not force-feed your code** with immutability
  - Immutability is very **intolerant of entangled design**, it will bite really hard
  - Immutability makes **working with associations more difficult** (bidirectional one-to-many and many-to-many) and odd for many people

Gateway to Functional Programming

- With immutability, **extracting** or **inlining** an expression **does not change the meaning** of the program
  - This is called **referential transparency** 😮
  - Fundamental property of **functional programming**
- FP is programming with **pure functions** 😇
  - **Deterministic**: same arguments implies same result
  - **Total**: result always available for arguments
  - **Pure**: no side-effects
- But how do we do with **I/O**?
  - Season finale cliffhanger... 😧