Functional constructs with java8 (and haskell)

Goal

- avoid confusing terms
- generate more question than answers
- notice similarities in the patterns between extremely different languages (the knowledge is transferable!)
- be frustrated by the limitation of java

Overview

- base of fp
- functions as a first class citizen
  - as a parameter
  - composition
- chaining functions
  - error handling

Required: few concepts

We need functions, immutable data structures and types

Yep. That much.

Functions are first class objects

First-class and Higher-order functions

- java :
  - java.util.Function<T, R>
  - Function<Integer, Integer> fun = (n) -> n*2;

  - ☑ can be passed as parameter to another function
  - ☑ can be returned from a function
  - ☑ can be stored in a variable/field
  - See Example1.higherOrder

First-class and Higher-order functions

Haskell:

- t -> r
- let fun n = n * 2
- fun has type (::) : **fun :: Num a => a -> a**
  - Simple interpretation
    - a must be an instance of Num (e.g: Integer, …)
    - the function has a parameter of type a and return the same type

Function

Every function in Haskell officially only takes one parameter. Let’s see if we can do the same in Java and why it’s interesting.

- modularity, composition, partial application
- Example2.curryExample

Tools: function composition

- it’s our main “design pattern”
- build a pipeline, the types will guide you

- Function<Apple, SlicedApple> f1;
- Function<SlicedApple, ApplePie> f2;
- f2.compose(f1) -> Function<Apple, ApplePie>
- f1.andThen(f2) -> Function<Apple, ApplePie>

Types for modelling

- Types represent constraints on input and output
- Avoid passing primitive types for declaring our intent more clearly
- Unfortunate in java: type declaration is not compact and has a cost :(
- (types are a form of proof)

Function design

- strive for purity: no side effects
- use descriptive types
- from simple to complex use cases, composition is the way to go

COMPOSED FUNCTIONS

FUNCTIONS

Lists/Stream

Important abstraction:

- see Stream in java (examples will be skipped)
- list in haskell

Tools: map/fmap

Wrapped types (Optional<T>, Stream<T>) provide a map/flatMap function, for lifting a “normal” function inside the container.

Examples: Example3.map

Tools: reduce/fold

Reducing/Folding is one of the main operation available on a collection.

Map can be built from reduce (not beautiful in java)

See Example04.reduce

Chaining functions and error handling

- the best case are functions that return only one “type” of result: no failure

- the external world is full of possible failure
- expose the type of error with a… type

Optional / Maybe

- Function<Integer, Optional<Integer>>
  - from the type signature it’s clear that something can fail

- Now we have a small problem, how can we chain them without creating a pyramid of checks? See Example5

Result type: when we care about the errors

Not present out of the box in java: comprehensible choice, the type system will be really hard on us here.

We want “2 types of return”

- the value
- the error

Result type

Partially based on the Either type, but simplified to the core for a “decent” experience with java

See Example6

monad

Few java libraries for “functional style” programming and others links

- http://javaslang.com/
- http://www.functionaljava.org/

- http://learnyouahaskell.com/chapters
- https://www.haskell.org/hoogle/

Questions ? :D

- preemptive answers (personal opinions):
  - java need: local type inference, sum types and pattern matching
  - use scala (?)
  - I’m not a fan of the scala type system
  - Haskell laziness is a blessing and a curse
  - checked exceptions are a good idea, but not how java has implemented (see effect systems)