FP Patterns

Monday, March 26, 2018 7:46 PM

Single Abstract Method (SAM) interface

- `@FunctionalInterface`
  - Compiler will alert when you try to add second abstract method to interface
- Lambda expression can be used to represent anonymous classes that implements any interface with only one abstract method

Functor

- Category theory: takes in function, returns functor (can return itself)

```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

- OO concept: variable wrapped in class in some context
- Lambda expression used for cross-abstraction barrier manipulation
- `LambdaList, InfiniteList, Stream`

Functor Laws

- Passing in identity function should not change functor
  - `x -> x`
- Composite functions can be called separately

Monad

- Takes in function (that returns monad) and returns monad
  - Takes in function that returns itself
- `E.g. Stream, with flatMap and Optional`

Monad Laws

- `of`
  - Wraps objects into monad
- Left identity law
  - `Monad.of(x).flatMap(f) == f(x)`
- Right identity law
  - `monad.flatMap(x -> Monad.of(x)) == monad`
- flatMap is associative

  `monad.flatMap(f).flatMap(g) == monad.flatMap(x -> f(x).flatMap(g))`

Implementing Strategy/ Policy

- Each class encapsulates a different way of achieving the same thing
- OOP/ FP better than imperative programming (switch statements)

`Runnable`

- Takes in no argument, returns nothing

Observer Pattern

- Observers/ dependents called by subject

Exercises

1. `createSnippet(TextShortener::shorten);`
    - @FunctionalInterface indicates that it has a Single Abstract Method (SAM)
2. Composite functions can be called separately