Design Constructs: Part II

Functional Programming

- Existed before OOP.
- Gaining importance/popularity now. Almost every language supports that.
- Take an application and decompose into a series of operations through functions.
- Functions are considered ‘first class citizens’. In OOP, objects are considered ‘first class citizens’.
- Eg. Haskell. `foldl` is a Higher-Order function, which takes as an input another function `add`. `foldl` is a reduce function.

```
-- Haskell

double value = value * 2
add op1 op2 = op1 + op2

main = do
    print(filter even [1..10])
    print(map double [1..10])
    print(map double (filter even [0..10]))
    print(foldl add 0 [1..10])
    print(foldl add 0 (map double (filter even [0..10])))
    -- another way of writing the same as above
    print(foldl add 0 . map double . filter even $ [0..10])
```

- Eg. Java: using `forEach`. `forEach` is an internal iterator — a higher-order function, because it accepts another function as a parameter. Other functions are `filter`, `map`, and `reduce`.
- Scala, Ruby, Groovy to an extent, all do a good job combining OOP & FP. Java also performs this ‘marriage’, where we can intermix this OOP and FP style of programming.

- Function Composition → taking the results of one function and feeding it into another and so on.

Lazy Evaluation

- Supported by Java, Haskell, etc.
- Applicative order vs Normal order.

```
-- Haskell

greetKid = print("Hello kiddo")
greetAdult = print("Howdy")

greet age greetKidfn greetAdultfn =
    if age < 15
    then greetKidfn
    else greetAdultfn

main = do
    greet 13 greetKid greetAdult --greetKid gets eval. greetAdul
-- lazily evaluated greetKid
```

- Create a prototype — have it evaluated by a mentor/colleague/teacher/groupmate. Don’t code in isolation.
- Put away emotions. Don’t go around assuming your design is perfect. Ask others to review it, ask them how you can improve your design.
- Incidental vs Accidental Complexity. Former comes from the problem/domain/application. Latter comes from the solution space. FP has less Accidental complexity compared to incidental style of programming.