Notes 7: functional programing

# CH 11 From Scott

Issues like binding times, names and scope still apply.

Why Functional Programming
“Imperative languages have shared mutating values”

1. A change in one place can effect others
2. 2. Concurrency

Scott “One can write in a function style in many imperative languages, and many functional languages include imperative features ....”

Common features of functional languages
Per Scott: “heavy use of polymorphism “

Remember “polymorphism”

Ad hoc polymorphism, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

There are two subkinds of ad hoc polymorphism.
Overloading refers to ad hoc polymorphism in which the language's compiler or

interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at compile time based on the declared types of the entities. They exhibit early binding.

Subtyping (also known as subtype polymorphism, inclusion polymorphism, or polymorphism by inheritance) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits late binding.

The object-oriented programming community often refers to inheritance- based subtype polymorphism as simply polymorphism.

Parametric polymorphism, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

The object oriented programming community often calls this type of polymorphism generics or generic programming. The functional programming community often calls this simply polymorphism.

FROM https://john.cs.olemiss.edu/~hcc/csci450/2016fall/notes/Fundamentals/ Polymorphism.html

IT IS PARAMETRIC POLYMORPHISM THAT FP USES. Claim:

Polymorphism offers the following advantages –
It helps the programmer to reuse the codes, i.e., classes once written, tested and

implemented can be reused as required. Saves a lot of time.
Single variable can be used to store multiple data types.
Easy to debug the codes.

https://www.tutorialspoint.com/functional_programming/functional_programming_polymorphism.htm

NOTE THIS IS CONTRADICTORY TO CLAIMS MADE IN STRONGLY TYPED LANGUAGES

Back to Scott and features of functional programming languages

1. lists (rather than arrays)

2. RECURSION
3. Lots of temporary variables so garbage collection first class functions
4. “structured function returns”

5. immutable values
6. functions are ONLY dependent on their arguments functions ONLY effect is its return value

function programming and top down thinking
“top down programming” is what you have been taught.
start with statement of problem, design classes, design function interfaces, write ... Linked to a method of software development “waterfall”
functional programming is more naturally “bottom up”.

start by writing a program to do one little piece of task. Make sure it works. write another function, that may use the result of first function, to do another

The final program ends up being a fairly simple assembly of the pieces.

You know it will work, because all of the pieces are easily and independently testable because each function depends only on its parameters

In functional programming you ALWAYS have something that works. May not do everything, but it does things correctly

Currying — a common programming technique in FP.

Suppose you have a function with 4 params. In one section of your code 3 of the 4 are always the same.

Currying means to create a new function with the three preset!

see curry/curry.go

# Java lambda give a functional like look

Lambda_lava/LambdaIntf.java

Lambda_lava/LambdaSort.java

Lambda_lava/LambdaEach.java

from https://www.baeldung.com/foreach-java

# 5. Foreach vs For-Loop in Java

From a simple point of view, both loops provide the same functionality: loop through elements in a collection.

**The main difference between them is that they are different iterators. The enhanced *for-loop* is an external iterator, *forEach* method is internal.**

## 5.1. Internal Iterator — *forEach*

This type of iterator manages the iteration in the background and leaves the programmer to just code what is meant to be done with the elements of the collection.

The iterator instead manages the iteration and makes sure to process the elements one-by-one.

```
names.forEach(name -> System.out.println(name));
```

In the *forEach* method above, we can see that the argument provided is a lambda expression. This means that the method only needs to know **what is to be done**, and all the work of iterating will be taken care of internally.

## 5.2. External Iterator — *for-loop*

External iterators mix **what** and **how** the loop is to be done.

*Enumerations*, *Iterators* and enhanced *for-loop* are all external iterators (remember the methods *iterator()*, *next()* or *hasNext()*?). In all these iterators, it's our job to specify how to perform iterations.

For example

```
for (String name : names) {
    System.out.println(name);
}
```

Although we are not explicitly invoking *hasNext()* or *next()* methods while iterating over the list, the underlying code that makes this iteration work uses these methods. This implies that the complexity of these operations is hidden from the programmer, but it still exists.

Point, the programmer has to do the work of iterating.

A quick look at Elixir