Functional Programming in Java

In Java, we pass objects to methods and return objects from methods. We can do the same with functions. That is, we can pass functions to methods and return functions from methods.

A higher-order function is a function that does at least one of the following:

1. takes one or more functions as arguments,
2. returns a function as its result.

Sometimes, it is important to execute some kind of operation on the items of a dataset, but this particular operation is defined at execution time. Sometimes there is just no way to know what action we want to take except that we need to do something at some particular point, but it is completely unknown.

**Example 1**

Consider a set of 4 functions, say Function A, Function B, Function C and Function D. The execution of task A requires one of the functions B, C or D to be executed but the exact name of the function is not known in advance. In this case, function A can get a function as an argument and perform a call to that function. The output will depend on which function is passed as the argument to function A.

**Example 2**

Consider a scenario where we need to build a function which takes a list and another function as it’s input, applies that function to each element of that list and returns the new list. In this case, higher order functions can be applied since otherwise, the argument in the list and the corresponding function to be executed has to be mapped explicitly.

```
let x = [1..100]
for each item in x do
    print(item)
```

Here, we are printing the each number in the list x by calling the function print. What about if we need to do something else with each item? Do we need to implement all possible cases that we think we will use?

```
let Function1 (f, list) =
    for each item in list do
        f(item)
```

Here, **Function1** is a higher order function because it received a function **f** as an argument, which will be executed on each item from list. Here, the function f is known in run time.

**Interfaces in Java which support functional programming**

**Consumer**

```
public interface Consumer<T> {
    void accept(T t);
    ....
}
```

A consumer accepts an input argument but does not return any output. A consumer is called through the accept() method.

**Supplier**

```
public interface Supplier<T> {
    T get();
}
```

A supplier does not accept any input but produces an output. The output from a supplier is obtained by calling its get() method.

**Function**

```
public interface Function<T, R> {
    R apply(T t);
    ...
}
```

A function accepts an input argument and produces an output. It is executed by calling its apply() method.

**Code snippet for Phone Directory - search scenario**

```
Class PhoneDirectory{

Map<String,String> phoneDirectory; /*A map is created between names
and numbers. It is assumed that a name can have only one phone
number corresponding to it.*/

//The following function receives a string as an argument and returns a
//string.
Function<String,String> search = (Name)->{
        String phoneNumber = PhoneDirectory.get(Name);
        return phoneNumber;
};

public static void main(String [] args){
        phoneDirectory = new HashMap<String,String>();
        phone.Directory.push(“Ramesh”,”98654432”);
        phone.Directory.push(“Suresh”,”98654433”);
        Double phone_number = search.apply(“Ramesh”);
 /*The number corresponding to the name Ramesh is retrieved from the
phone directory.*/
        }
}
```