Higher-order function

A higher-order function also known as functional form, functional or functor is a function that does at least one of the following

* Takes one or more functions as parameter
* Returns a function

In the first post [Lambda Calculus](http://data-structure-learning.blogspot.com/2015/05/lambda-calculas.html) we discussed two different simplifications that we can chain functions inside functions.

* x → (y →(some formulae))(parameter1)) (parameter2)
* More precisely it can be written as (T1 → T2) →T3.

Higher order function can be achieved by using a Function<T, R> interface.

Function<T, R> interface accepts the input parameter with type T and returns result R i.e. result of the function. Function can be used to transform an input to another value.

Let us take a simple example to use the Function interface with Predicate interface in filter method.

Let us first write a Predicate.

Predicate<String> lengthCheck = **new** Predicate<String>() {

**public** **boolean** test(String str){

**return** str.length() > length;

}

};

This predicate is used to test for String length greater than some length passed as parameter.

Now let us take length argument as type Integer and use in Function interface.

Function<Integer, Predicate<String>> lengthChecker

Integer is type for parameter length

Predicate<String> is the return type of the Function.

Now, there is a method apply(T t) in Function interface. It is used to apply this function to the given argument. The return type of apply(T t) method is result itself.So we can override the apply(T t) method and write predicate inside it and return Predicate itself.

@Override

**public** Predicate<String> apply(Integer length) {

Predicate<String> lengthCheck = **new** Predicate<String>() {

**public** **boolean** test(String str) {

**return** str.length() > length;

}

};

**return** lengthCheck;

}

Now let us write entire Function.

**final** Function<Integer, Predicate<String>> lengthChecker = **new** Function<Integer, Predicate<String>>() {

@Override

**public** Predicate<String> apply(Integer length) {

Predicate<String> lengthCheck = **new** Predicate<String>() {

**public** **boolean** test(String str) {

**return** str.length() > length;

}

};

**return** lengthCheck;

}

};

Input is

List<String> names=**new** ArrayList<String>(Arrays.*asList*("Robb","Ramsey","Ned","Sansa"));

And we call this function as

**long** namesCount = names.stream().filter(lengthChecker.apply(4)).count();

apply(T t) is overridden method and it takes Integer as input. We passed 4 as input to apply(T t) method.

Then output will be **2** because there are **2** names ("Ramsey","Sansa") which have length greater than 4. Dear game of thrones fans, don’t get angry for output. ☺

We can achieve this thing using Predicate interface too why did we wrote this complex syntax for Function interface. There are high chances to make mistake in this syntax. We will see concise version in a moment.

There are few difference between Predicate<T> interface and Function<T, R> interface. Click here to read difference between them and why we used Function<T, R> interface.

Now let us use some lambda operator to make this verbose code concise.

**final** Function<Integer, Predicate<String>> lengthChecker = **new** Function<Integer, Predicate<String>>() {

@Override

**public** Predicate<String> apply(Integer length) {

Predicate<String> lengthCheck = **new** Predicate<String>() {

**public** **boolean** test(String str) {

**return** str.length() > length;

}

};

**return** lengthCheck;

}

};

We know that Function and Predicate interface are Functional Interfaces so we can use lambda operator for both of them hence reducing the code significantly in size.

**final** Function<Integer, Predicate<String>> lengthChecker = (Integer length) -> {

Predicate<String> lengthCheck = **new** Predicate<String>() {

**public** **boolean** test(String str){

**return** str.length() > length;

}

};

**return** lengthCheck;

};

So in above step we just removed the apply(T t) method as we know that Function<T, R> is Functional Interface.

Next we will remove test(T t) method as Predicate<T> is also Functional Interface.

**final** Function<Integer, Predicate<String>> lengthChecker = (Integer length) -> {

Predicate<String> lengthCheck = str -> str.length() > length;

**return** lengthCheck;

};

Next we will reduce this code as we don’t need to write Predicate<String> lengthCheck explicity.

Function<Integer, Predicate<String>> lengthChecker = length -> str -> str.length() > length;

The above line of code is more concise and easy to understand than to write anonymous inner class syntax.