his is a term you may not have heard. Although the term isn’t used often, these kinds of functions are common. We use higher-order functions all the time when we use streams.

Higher-order functions are functions that either accept other functions as arguments, or return a function as a result.

The result of one higher-order function can be used as the input to another higher-order function. Any time we pass a lambda expression to a method, that method is a higher-order function.

Java 8 supports higher-order functions, along with lambda expressions and functional interfaces. Stream processing uses lambda expressions and higher-order functions extensively.

We create Predicate variables and assign lambda expressions to them

A Predicate represents a function that takes one argument and returns a boolean value. The functional method is boolean test(Object). So the interface looks like this (omitting default and static methods):

@FunctionalInterface

public interface Predicate {

public boolean test(T t);

}

We create Predicate variables and assign lambda expressions to them as follows:

Predicate pred1 = e -> e.getAge() < 65;

Predicate pred2 = e -> e.getSalary() > 10000.0;

*// using a default method of Predicate*

Predicate pred3 = pred1.and(pred2);

But what about the test() method? We never create one. And we do not call it in the normal course of programming, for example, when processing streams.

*// create and populate a list with employee objects...*

List list = new ArrayList<>();

...

*// use a stream to process the list*

list.stream().filter(p1).filter(p2).forEach(

e -> System.out.println(e));

This filter() method is a higher-order function that takes a Predicate as a parameter. The filter() method is applied to each element to check if it should be included in the stream. Inside the filter() method, the test() method of the Predicate is called, and the lambda expression is executed.

**Writing our own higher-order function**

Let’s try it out by creating our own higher-order function that takes a Predicate:

public void runPredicate(Employee e, Predicate p) {

if (p.test(e) == true)

System.out.println("True");

else

System.out.println("False");

}

We can call the higher-order function as follows:

Employee e1 = new Employee("Harriet", 31,

Person.FEMALE, 30\_000.0);

Predicate pred1 = e -> e.getAge() < 65;

Predicate pred2 = e -> e.getSalary() > 10000.0;

*// using a default method of Predicate*

Predicate pred3 = pred1.and(pred2);

runPredicate(e1, pred1);

runPredicate(e1, pred2);

runPredicate(e1, pred3);

There it is! Simple as that!

--------------------------

Higher-order function is an essential part of the functional programming paradigm. We must have defined a lot of functions in any language where we pass either primitive types or an object as an argument and returns the same. These are normal functions or methods.

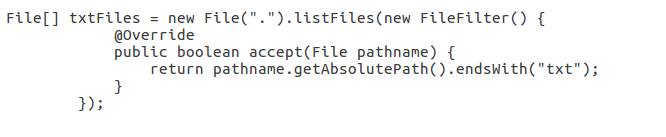
Then comes first-class functions. First-class functions are the functions that are treated as values. It means, those functions can be assigned to a variable and can be passed around as an argument.

And finally, comes higher-order functions. These are the functions that either takes one or more function as an argument or returns the function as a result.

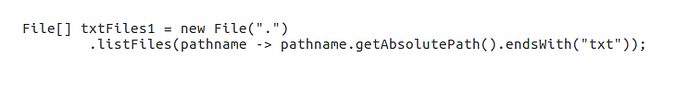
The two things (First class function and higher-order function) are closely related, as it’s hard to imagine a language with first-class functions that would not also support higher-order functions, and conversely a language with higher-order functions but without first-class function support.

Before Java8, we used to pass a function to a function with the help of anonymous inner classes, but now we do that with the help of Lambdas.

**Before Java 8:**



**After Java 8:**



In the above example,

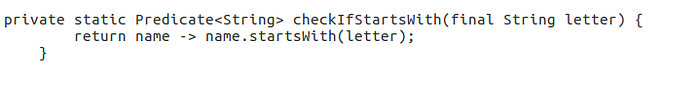
we are passing **pathname -> pathname.getAbsolute().endsWith(“txt”)** as an argument to **listFiles().**

Generally, a function has a body, a name, a parameter list, and a return type. The passed function here has a parameter list followed by an arrow ( -> ), and then the short body. The type of the parameter may be inferred by the Java compiler here and the return type is implicit. This function is anonymous; it has no name.

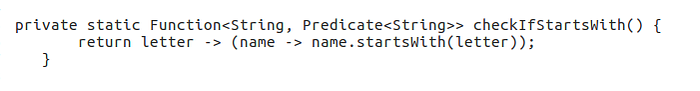
Rather than referring to these as anonymous functions, we call them lambda expressions.

Let’s understand the higher-order function in deep with some code examples:

**Returns function as a result:**



In the above example, checkIfStartsWith() returns Predicate.



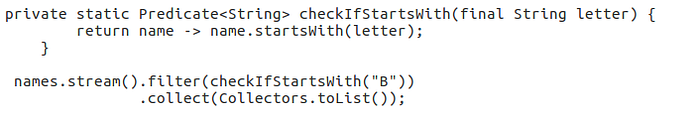
In the above example, checkIfStartsWith() returns Function which takes 1 argument and returns the Predicate.

bifunction

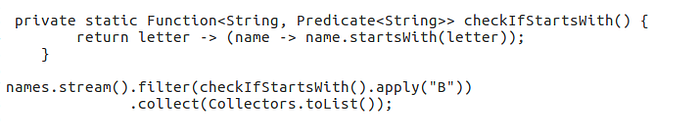
In the above example, getList() returns BiFunction which takes 2 arguments and returns the List.

Predicate, Function, and BiFunction are the functional interface that is used to write Lamda expression in the above examples.

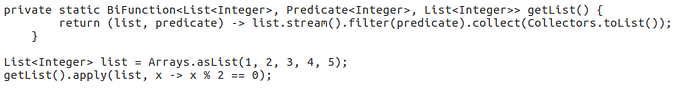
**Take the function as an argument:**



In the above example, filter() is the higher-order function which takes a Predicate as an argument.



In the above example, filter() is the higher-order function that takes Function as an argument.



In the above example, getList() is the higher-order function that takes Predicate as an argument.

We have taken the examples only for 3 different Functional Interfaces, but there are more predefined functional interfaces such as Supplier, Consumer, etc and even we can define our own custom Functional interface and can use that to define Lambda expression.

So that is all about higher-order function. I hope this blog will help you to understand the higher-order function in Java which is enabled in Java 8 with the help of Lamdas and Functional interfaces.

A higher order function is a function that either takes a function (method) as parameter, or returns a function after its execution. In this higher order tutorial I will show a few examples of higher order functions in Java.

## Sorting Collections

The first example of a higher order function is the Collections.sort() method which takes a Comparator as parameter. Here is an example:

List<String> list = new ArrayList<>();

list.add("One");

list.add("Abc");

list.add("BCD");

Collections.sort(list, (String a, String b) -> {

return a.compareTo(b);

});

System.out.println(list);

The Collection.sort() takes two parameters. The first parameter is a List and second parameter is a lambda (function). The lambda parameter is what makes Collections.sort() a higher order function.

### Sorting in Reverse Order

Here is another example of a higher order function. This time it is a function that returns another function as result. Here is the Java higher order function example:

Comparator<String> comparator = (String a, String b) -> {

return a.compareTo(b);

};

Comparator<String> comparatorReversed = comparator.reversed();

Collections.sort(list, comparatorReversed);

System.out.println(list);

This example first creates a Java lambda expression that implements the Comparator interface.

Second, the example calls the reversed() method on the Comparator lambda. The reversed() method returns a new Comparator lambda, which reverse the result returned by the first Comparator implementation. By "reversing" I mean that it simply returns -1 \* comparator.compare(a,b)

Because the reversed() method returns a lambda (function), the reversed() method is considered a higher order function.

Third, the example sorts the List of Strings using the Collections.sort() method.

-----------------