## What are Lambda Expressions?

Lambda expressions are another way of writing inline unnamed code block which performs a simple function. Not a direct analogy but if you recall anonymous classes in Java they are a way of writing inline classes. On similar lines Lambda expressions are a way to write method body inline.

They can be considered as anonymous methods, there is no need to formally define these methods but the code in the lambda expression is the body of the method.

What is Functional Interface?

A functional interface is any interface which contains only one abstract method. A functional interface can have one or more default and static methods.

Also, there is an annotation in Java @FunctionalInterface which is used to annotate the functional interfaces. It is not a mandate for functional interfaces to carry this annotation.

There are many predefined functional interfaces in Java which belong to the package **java.util.function**

What is the significance of these interfaces?

These interfaces as we know declare a method which can be implemented and used for executing expressions as any other interface. But the power comes in conjunction to the [Lambda Expressions](http://techieme.in/introduction-to-lambda-expressions-java8/) we learnt in the previous post.

As the lambda expressions, are inline method definitions, it can serve as an implementation to functional interfaces.

There are primarily four types of functional interfaces defined in the language:

* Function
  + It represents a relation which transforms an input T into a result R.
  + The functional method is **R** **apply(T t).**
* Consumer
  + It represents a relation which consumes an input T and returns no output.
  + The functional method is **void accept(T t).**
* Predicate
  + It represents a relation which evaluates an input T and produces a boolean result.
  + The functional method is **boolean test(T t).**
* Supplier
  + It represents a supplier of results
  + The functional method is **T get().**

Apart from these methods there are certain default and static methods defined in these interfaces. Yes! you heard it right, you can define methods with bodies in Java 8.

There are other categorization of the Functional Interfaces based on the number of arguments:

* BiPredicate
* BiConsumer
* BiFunction

Each of these serve the same purpose, but accept two arguments. These interfaces can be specialized to get more specific functionalities like the BinaryOperator<T> interface. The apply function in this interface takes two arguments of the same type and the returned result is of the same type as well.

There are 40+ such interfaces which you can look up [here](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

How to use these interfaces?

**The Interface : Consumer**

Here is a code block to explain things in a simple way.

public static <T> void testConsumer(List<T> L, Consumer<T> P) {

for (T q : L) {

P.accept(q);

}

}

This is a method which expects a list of type T and a Consumer interface implementation. Then it iterates over the list and invokes the *accept* method on each element of the list as per the definition of Consumer interface.

The body of the *accept* method is the lambda expression which the caller passes to this method.

Now here is the code which invokes this method

public static void main(String[] args) {

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

L.add("Hello");

L.add("World");

testConsumer(L, j -> System.out.println(j));

}

The code passes the list to the *testConsumer* method and also passes a lambda expression which basically prints the argument passed to it.

When we run the program, it would print  
*Hello*  
*World*

**The Interface : Predicate**

The below code block is for the interface Predicate

public static <T> void testPredicate(List<T> L, Predicate<T> P) {

for (T q : L) {

System.out.println(P.test(q));

}

}

This is a method which expects a list of type T and a Predicate interface implementation. Then it iterates over the list and invokes the *test* method to each element of the list as per the definition of Predicate interface.

The body of the *test* method is the lambda expression which the caller passes to this method.

Now here is the code which invokes this method

public static void main(String[] args) {

List<Integer> L = new ArrayList<Integer>();

L.add(5);

L.add(6);

testPredicate(L, j -> j == 5);

}

The code in the main method passes the list to the *testPredicate* method and also passes a lambda expression which basically tests if the argument is equal to the number 5.

When we run the program, it would print

*true  
false*

**The Interface : Supplier**

The below code explains how to use Supplier

public static void testSupplier(Supplier<String> S) {

System.out.println(S.get()+" prasad");

}

This is a method which expects a Supplier interface implementation. Then it gets the element from the Supplier and  appends a surname to the end of the result returned from the Supplier.

The body of the *get*method is the lambda expression which the caller passes to this method.

Now here is the code which invokes this method

public static void main(String[] args) {

String s = "Hello my name is dharmendra";

testSupplier(() -> s);

}

When we run the program, it would print

*true  
false*

**The Interface : Function**

Here is the code to use the Function interface

public static <T, R> void testFunction(T t, Function<T, R> F) {

System.out.println(F.apply(t));

}

This is a method which expects a Function interface implementation and an element of type T. Then it invokes the apply method on the argument t and returns the result.

The body of the applymethod is the lambda expression which is passed by the caller to this method.

Now here is the code which invokes this method

public static void main(String[] args) {

Integer t = 9;

testFunction(t, q -> System.out.println("The square of "+ q+" is " + (q \* q));

}

When we run the program we get the following output:

The square of 9 is 81

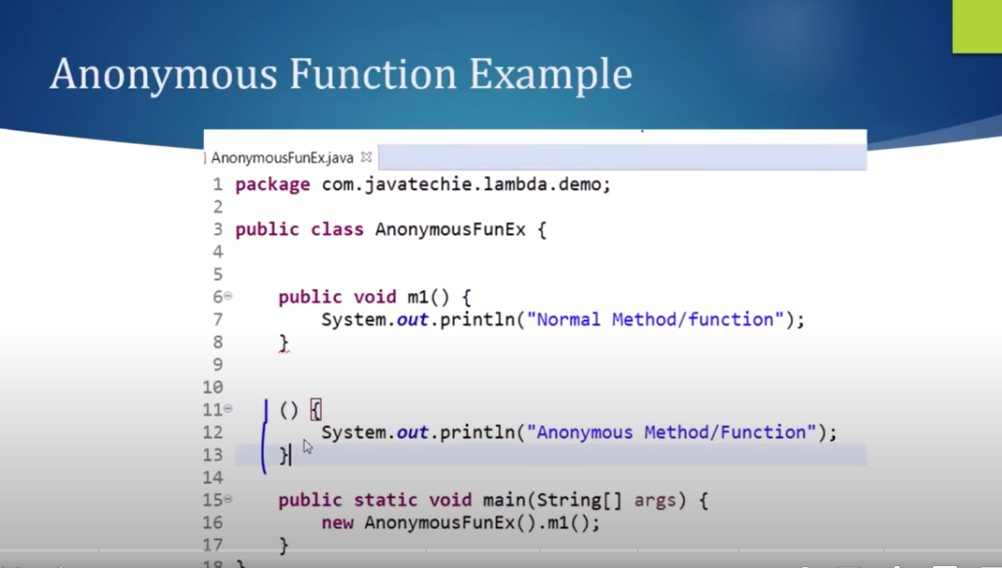
Conclusion

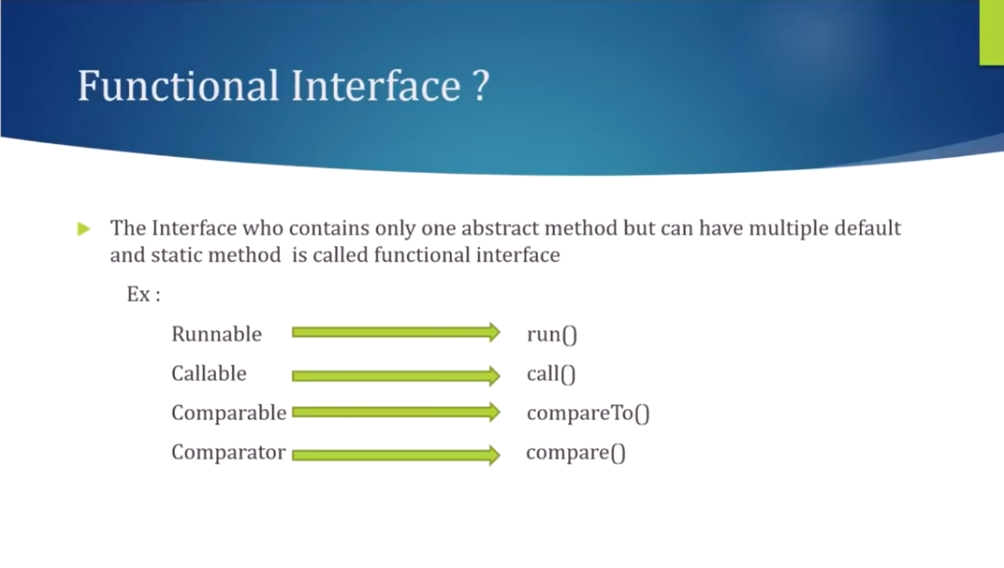
Here we learnt the functional features introduced in Java 8. The interfaces explained above is the core of the functional feature. There are many more interfaces around 40+ defined by Java.

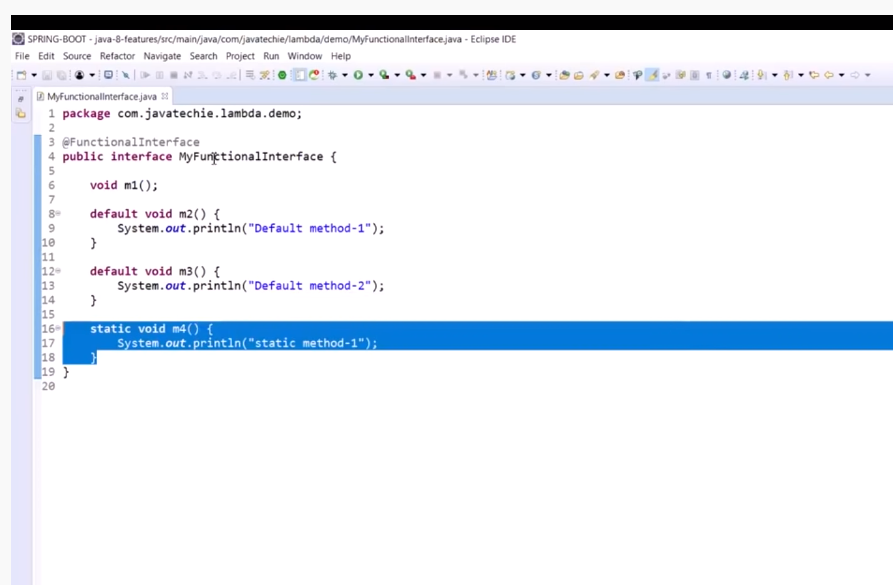
But once you understand the above four, rest all is just specialization or a two-arity form of these interfaces.

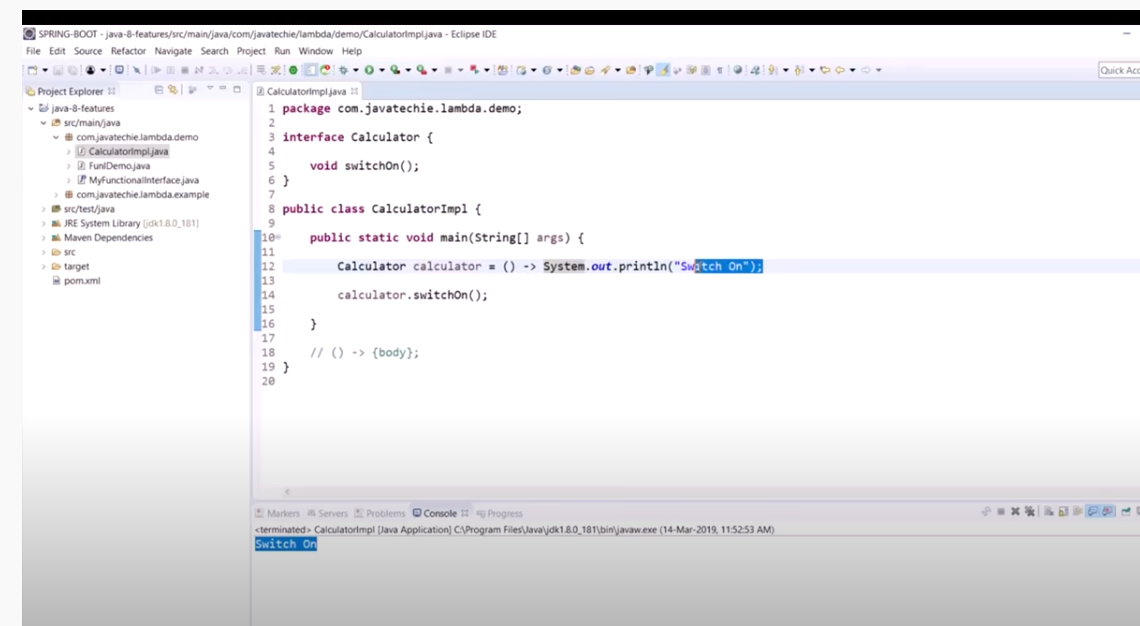
Of course there are many different ways of using these functional interfaces, the ones shown above are the basics. Going forward I will write different ways of using these interfaces.

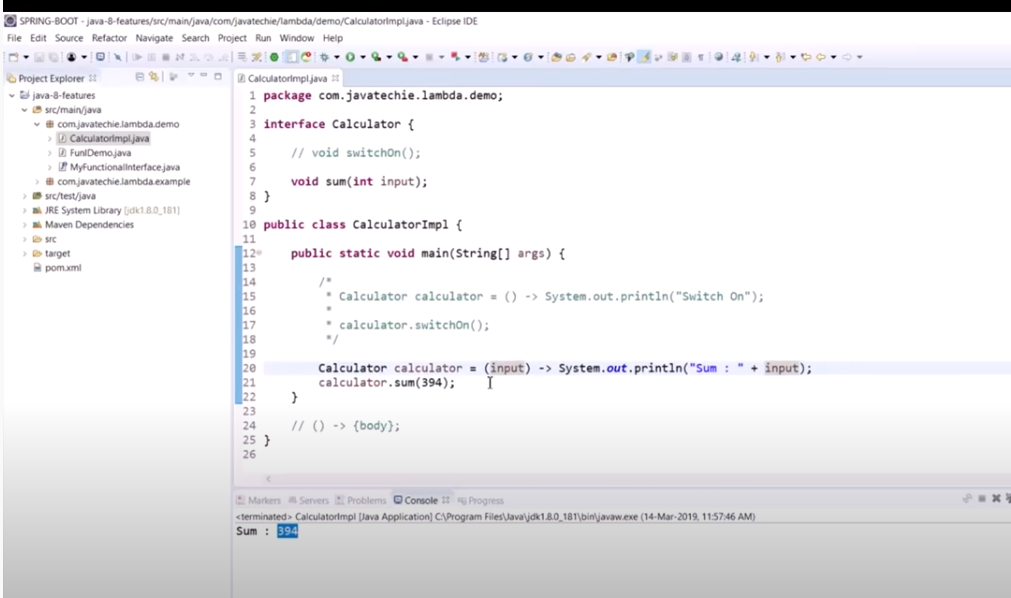
https://techieme.in/introduction-to-functions-java8/

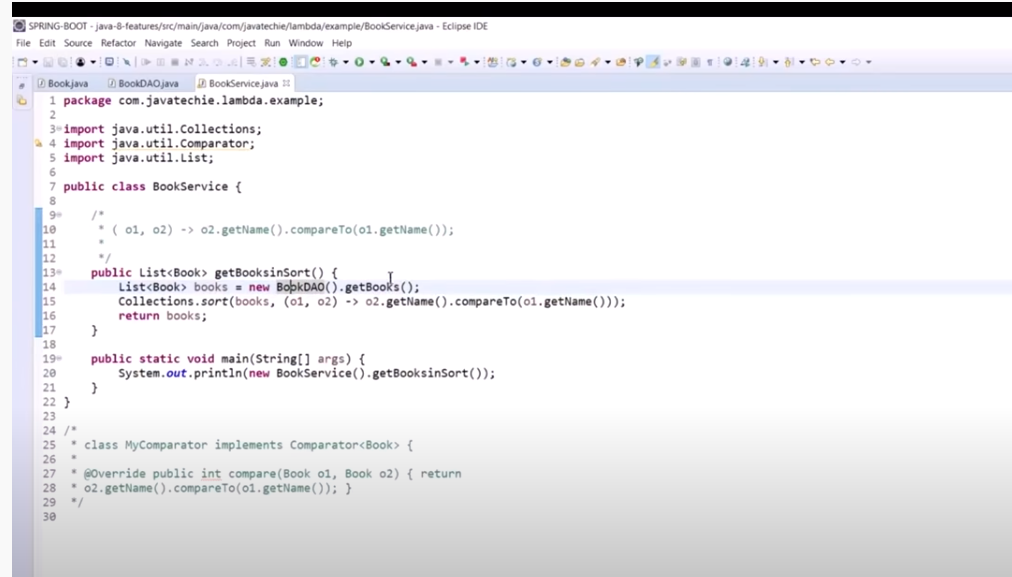




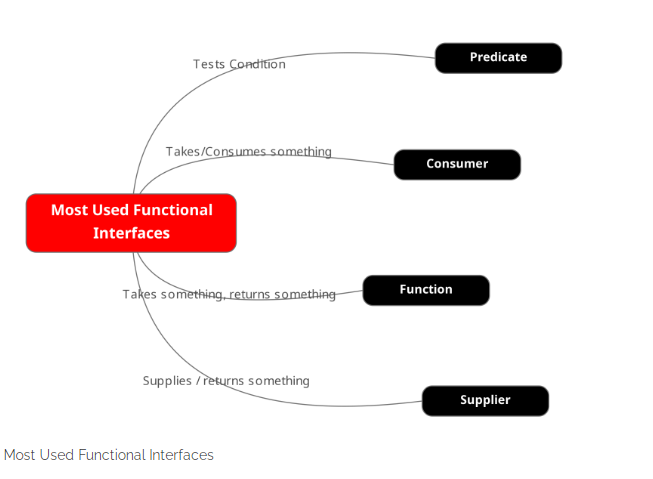
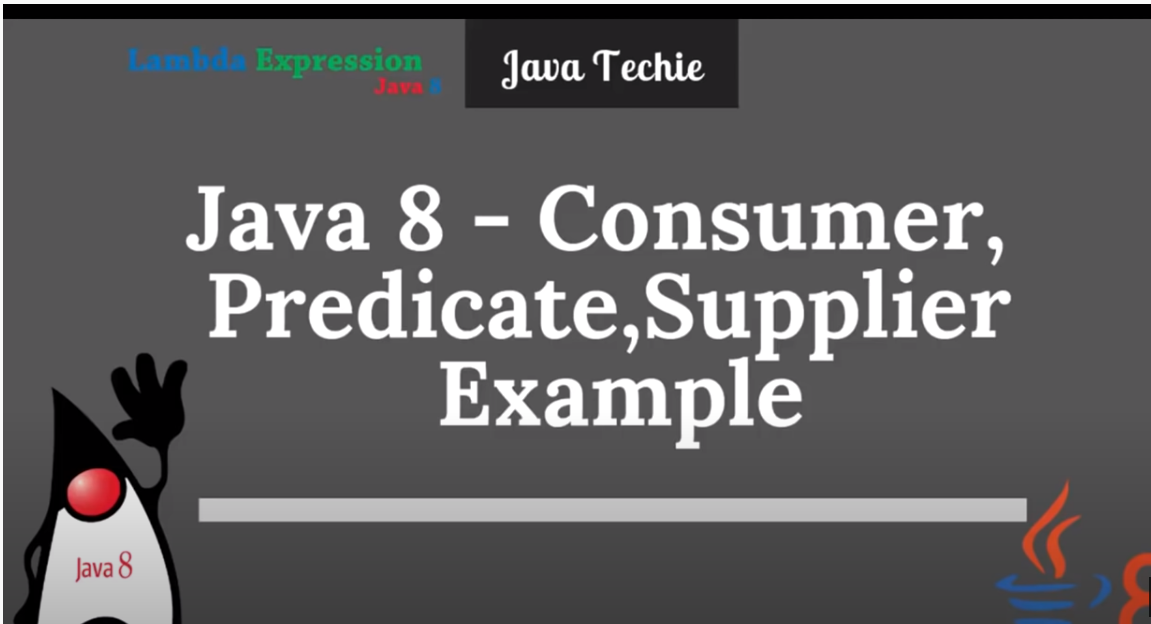








2nd Session



https://basicsstrong.com/understanding-predefined-functional-interfaces-in-java/

