Java Functional Programming

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Here are just a few of the key benefits to using lambda expressions in Java:

* Conciseness
* Reduction in code bloat
* Readability
* Elimination of shadow variables
* Encouragement of functional programming
* Code reuse
* Enhanced iterative syntax
* Simplified variable scope
* Less boilerplate codes
* JAR file size reductions
* Parallel processing opportunities

Lambda Expression is an Anonymous function.

1. No Access modifier

2. No return Type

3. No function name

Normal function to Lambda Expression

**public** **void** m (**int** a) {

**return** a\*a;

}

(**int** a)->{**return** (a\*a);}; // if we have return statement we must use {};

(**int** a)-> a\*a; // no return statement and single line of code {} is optional

(a) -> a\*a; // single variable, type is optional as so as ();

a -> a\*a;

Test Different style

n-> **return** n\*n; // Not valid, we must use {} with return statement

n->{**return** n\*n;}; //Valid

n->{ **return** n\*n}; //Invalid missing; at the end of the return statement

n-> { n \*n;}; // Invalid {} are not required

n->n\*n; // Valid

How to call (Invoke) Lambda Expression.

Lambda Expression can be invoked only through functional interfaces.

Interface contains only Abstract Method (Signature).

An Interface with only one Abstract Method called *functional interface*.

Functional Interface Concept is an Interface with a following conditions:

Must contains only single Abstract method. (SAM)

Any number of default methods

Any number of static methods

Some predefined functional interfaces and their Abstract method used In java Classes

|  |  |
| --- | --- |
| Interface | Abs Method |
| Runnable | run() |
| Callable | call() |
| Comparable | camperTo() |
| ActionListener | actionPerformed() |

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface Runnable

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface Callable<V>

A task that returns a result and may throw an exception. Implementors define a single method with no arguments called call.

The Callable interface is similar to [Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html), in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by [Collections.sort](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-) (and [Arrays.sort](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-java.lang.Object:A-)). Objects that implement this interface can be used as keys in a [sorted map](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html) or as elements in a [sorted set](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html), without the need to specify a [comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html).

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface ActionListener

extends [EventListener](https://docs.oracle.com/javase/7/docs/api/java/util/EventListener.html)

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

**Define a Functional Interface**

@FunctionalInterface // Java >7

**interface** Cab

{

**public** **void** bookCab(); //SAM (single abstract method)

}

We can Implement bookCap() via a class:

**class** Ola **implements** Cab

{

**public** **void** bookCab() {

System.***out***.println("ola cab taxi is booked");

}

}

**public** **class** Lambda\_demo {

**public** **static** **void** main(String [] args) {

Cab b = **new** Ola();

Ola a = **new** Ola();

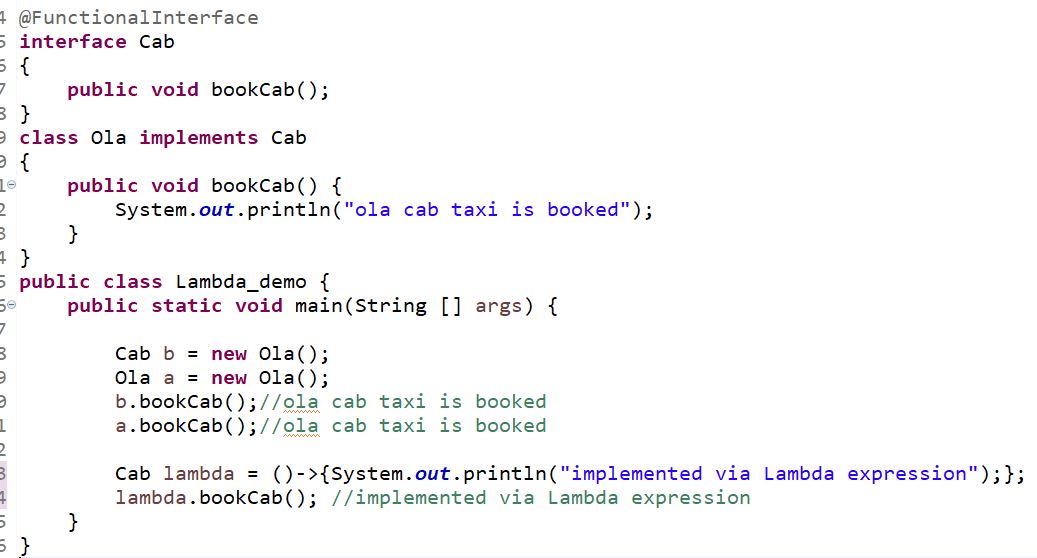
b.bookCab();//ola cab taxi is booked

a.bookCab();//ola cab taxi is booked

}}

since Cab is Functional Interface(SAM) we can use Lambda Expression to implement bookCap() method.

Cab lambda = ()->{System.***out***.println("implemented via Lambda expression");};

lambda.bookCab(); //implemented via Lambda expression

@FunctionalInterface

**interface** Cab

{

**public** **void** bookCab(String source , String des);

}

**public** **class** Lambda\_demo {

**public** **static** **void** main(String [] args) {

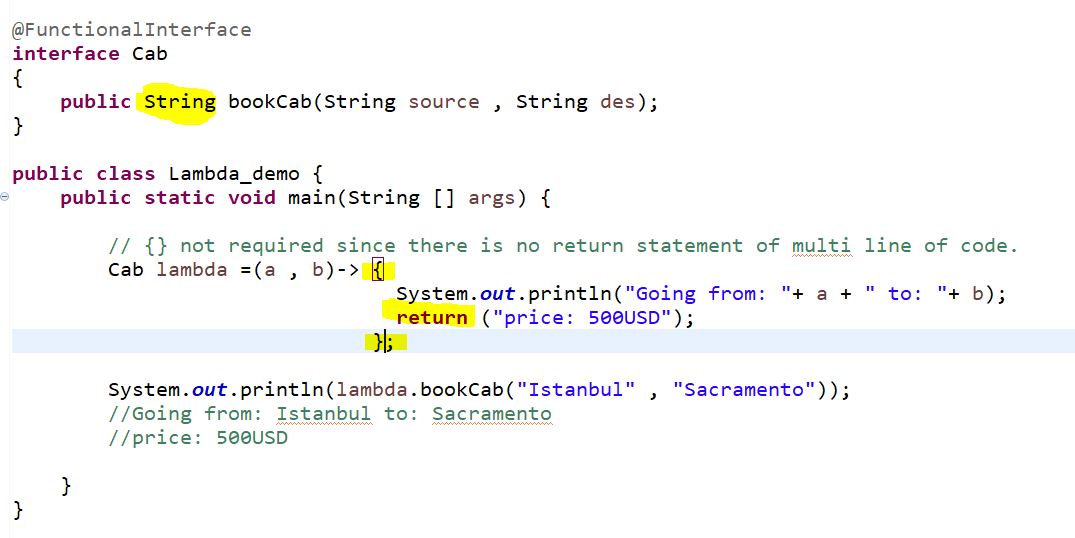
// {} not required since there is no return statement of multi-line of code.

Cab lambda =(a , b)-> System.***out***.println("Going from: "+ a + " to: "+ b);

lambda.bookCab("Istanbul", "Sacramento"); //Going from: Istanbul to: Sacramento.

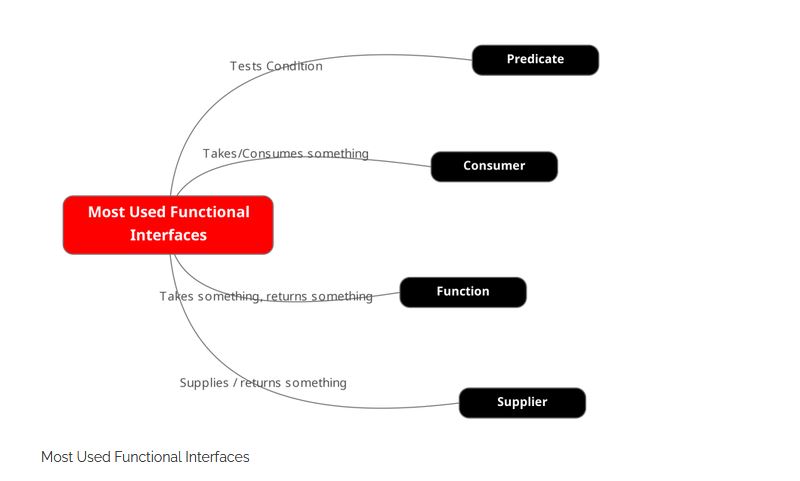
}

}



In the above example we learned how to code Lambda expression and how to call it using Functional Interface. Now It is time to go over java 8 predefined functional interfaces in **java.utill.function** and their usage.

**Understanding Predefined Functional Interfaces in JAVA**



For this JAVA has already provided most general reusable signatures in form of predefined interfaces and these are added to the package **java.util.function**. This package contains already defined GENERIC functional interfaces that can be used readily for invoking Lambda.

**Java Provides a function interface which allows us to define functions as Variables.**

## Interface Predicate<T>

* Type Parameters:

T - the type of the input to the predicate

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface Predicate<T>

Represents a predicate (Boolean-valued function) of one argument.

This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).

## Interface Function<T,R>

* Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

All Known Subinterfaces:

[UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html)<T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface Function<T,R>

Represents a function that accepts one argument and produces a result.

This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-).

package functional\_Programming\_with\_Java;

import java.util.function.Function;

public class The\_Function\_Interface {

protected static class Mymath{

public static Integer triple(Integer x) {

return x\*3;

}

public static void main(String[] args) {

// inputType , returnType

Function< Integer , Integer > myTriple = Mymath::*triple*;

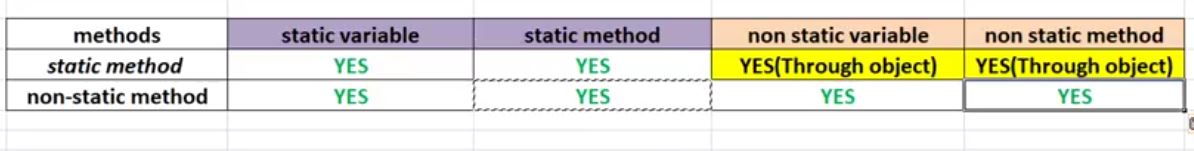
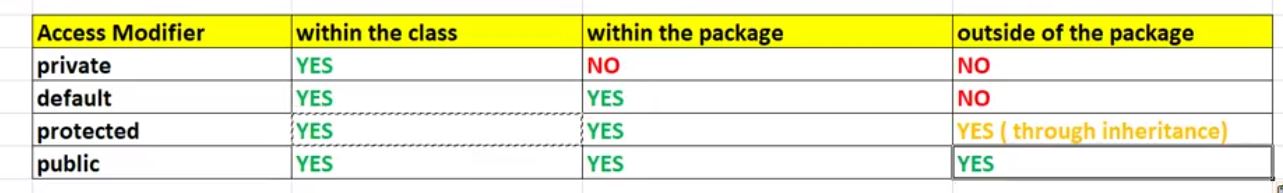
Integer result = myTriple.apply(5);

System.***out***.println(result); //15

}

}

}

Multiple Inheritance using Interface

Class to class we use extends

Interface to class we use implements

Interface to Interface we use extends

1- Final keyword can be used for variable and method & class.

Final keyword for variable makes it constant.

Final keyword for method: method cannot be override in the child class.

Final keyword for class: class cannot be extended