**Functional & Reactive Programming**

Lambda – is a function without a name, so called Anonymous function.

**public** **class** LambdaExample1 {

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

Thread t = **new** Thread(() -> System.***out***.println("Thread Executed"));

t.start();

}

}

We can get the above Lambda function from the below.

**public** **class** LambdaExample1 {

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

Thread t = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Thread Executed");

}});

t.start();

}

}

Because the compiler is able to infer given the following:

* Runnable is a Functional interface which has an abstract method of run.
* The run method returns a void, so the implementation of the run method does not return a value. It simply prints a String.

**Functional Interfaces**

The basis of a Lambda function. A Functional Interface should have a single abstract method. The reason is that there is no ambiguity to the code that implements the method as to which abstract method is being implemented.

@FunctionalInterface

**public** **interface** MyFuncInterface {

**public** **void** myMethod();

}

The @FunctionalInterface is not mandatory. It helps however, to warn you if the interface is modified in a way that it no longer is a Functional Interface. That would happen if another abstract method is defined.

**How to use the Functional Interface**

We define the MyFuncInterface and provide the implementation of the abstract method myMethodand we can then call the method.

**public** **class** FunctionalInterfaceDemo {

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

MyFuncInterface myFunc = () -> System.***out***.println("Hello");

myFunc.myMethod();

}

}

The following demonstrates how the method can be used as an argument. We can call the *onTheFly()* method passing the implementation of the abstract method because the *onTheFly()* expects a *MyFuncInterface* argument.

**public** **class** FunctionalInterfaceDemo {

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

*onTheFly*(() -> System.***out***.println("Greetings"));

}

**public** **static** **void** onTheFly(MyFuncInterface behavior) {

behavior.myMethod();

}

}

Function can have

* Access Modifiers (Private, Public, protected etc)
* Return type
* Name of the function
* Parameter List and Parameter Type
* Method body
* Return statement (in case anything is returned)

A Lambda has

* A list of parameters
* An arrow -> which separates the list of parameters from the body of the lambda function.
* The body of the lambda

No Parameter No Return

**public** **void** myName() {

System.***out***.println("No Parameter no return");

}

With a Functional Interface defined as follows

@FunctionalInterface

**public** **interface** Name {

**void** myName();

}

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

Name name = () -> System.***out***.println("No Parameter no return");

name.myName();

}

Method with Parameter and No Return Type

@FunctionalInterface

**public** **interface** FirstnameLastname {

**void** myName(String firstname, String lastname);

}

FirstnameLastname fnameLname = (fname, lname) ->

System.***out***.println("Firstname: " + fname + " Lastname: " + lname);

fnameLname.myName("Samson", "Marikwa");

The type of the arguments can be guessed by the compiler using type inference. This is because the types are specified in the Functional Interface.