* Lambda expressions
* Functional Interfaces
* Method references (::)
* Default methods
* Streams
* Date API

**Lambda Expression:** It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

**Syntax:** FunctionalInterface reference= (parameters) -> function body

NOTE: We can pass multiple parameters and body statement in lambda expression. But we if add more then one statement in body the we have to make return as last statement.

It works only with functional interface.

**Functional Interfaces:** A interface having only one abstract method.

NOTE: @FunctionalInterface doesn’t allow to add more than one abstract method in the interface at compile time. So, its good practice to add @FunctionalInterface.

We can put static or default method inside the functional interface.

Inside the **java.util.function** we will find all the predefine functional interface. Like Predicate.

Core Functional predefine interfaces

* **Predicate:** If we need to test the condition like if condition then we will go through with Predicate interface. It will return the Boolean value.

Public interface Predicate<T> {

boolean test(T t)

}

EX:

Predicate<Integer> oddEvenPredicate=(Integer x)-> x%2==0?true:false;

oddEvenPredicate(38);

oddEvenPredicate(45);

Here, we are passing the int value and due to <Integer> it will affect the performance in case of autoboxing. So, if we sure the value will be only in integer then we should use IntPredicate instead of Predicate and pass “int” as a parameter.

Ex: IntPredicate oddEvenPredicate=(int x)-> x%2==0?true:false;

* **Consumer:** It will consume the data but not produce anything. Like if we have to pass some input and print it.

Public interface Consumer<T>{

void accept(T t)

}

EX:

Step 1: Consumer<Interger> mConsoleConsumer=(Integer x)-> System.out.println(x);

Step 2: printNumberList(Arrays.asList(10,15,20,30,40,50), mConsoleConsumer); // for console, same we will jest replace the consumer for file output.

Step 3: public void printNumberList(List<Integer> nos, Consumer consumer){

for(int i=0;nos.length();i++){

consumer.accept(nos.get(i));

}

}

Here, we are passing the consumer object to print the numbers. Suppose, in future if we need to print these number in a file then only, we have to create the new consumer for file. Nothing will change in logic of main implementation. Like:

**Consumer<Interger> mFileConsumer=(Integer x)-> writeToFile(x);**

* **Supplier:** It will not accept any value just return the values.

**Syntax:**

**public interface Supplier<T>{**

**T get();**

**}**

Ex:

Step 1: Supplier<T> supplier=()->random.nextInt(100);

Step 2: printGrade(supplier)

Step 3: printGrade(Supplier<T> supplier){

Integer marks= supplier.get();

// logic to find the grade using marks.

}

* **Function:** It will accept only one input and return only one output.

**Syntax:**

Public interface Function<T,R>{

R apply(T t);

}

Ex:

Function<Integer,String> function=(Integer marks)->marks>40? ”pass”:”failed”;

System.out.println(“Result=”+function.apply(50)) //pass

System.out.println(“Result=”+function.apply(30)) //failed

**Method references:** In method reference :: always return the Function object (it’s a type of functional interface).