Consumer Interface Java (8)

It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default and Then() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

|  |  |
| --- | --- |
| **Method** | **Description** |
| void accept(T t) | It performs this operation on the given argument. |
| default Consumer<T> andThen(Consumer<? super T> after) | It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed. |

**Consumer interface is the target for lambda expression.**

The Consumer Interface is a part of the java.util.function package which has been introduced since Java 8, to implement functional programming in Java. It represents a function which takes in one argument and produces a result. However these kind of functions don’t return any value.

Hence this functional interface which takes in one generic namely:-

T: denotes the type of the input argument to the operation

The lambda expression assigned to an object of Consumer type is used to define its accept() which eventually applies the given operation on its argument. Consumers are useful when it not needed to return any value as they are expected to operate via side-effects.

**Example –**

**package** default\_pack;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Consumer;

**public** **class** ConsumerInterfaceExample {

**public** **static** **int** doubled(**int** a) {

**return** a\*a;

}

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

// 1.0 Consumer interface

Consumer<List<Integer>>numbers = (n)->{

**for**(**int** i=0; i<n.size(); i++){

n.set(i,n.get(i)\*n.get(i));

}

};

// 2.0 Consumer Interface with static method reference....

Consumer<List<Integer>>display = n-> n.forEach(System.***out***::println);

System.***out***.println("Consumer Interface Example of accept() and default andThen()");

// Calls both methods back to back with accepted list....

numbers.andThen(display).accept(Arrays.*asList*(1,2,3,4,5));

Here in the consumer interface 1.0, we have instantiated (Created Object) of consumer interface numbers of type<List<Integer>>. Here what we are doing is, we are providing definition to the accept method of consumer interface, which takes in one para and does not return anything.

Here in the consumer interface 2.0, we have instantiated (created object) of consumer interface again and provided definition to the accept method of consumer interface.

Now user can directly invoke the assigned method definition as below –

* **display.accept(Arrays.asList(1,2,3,4,5));**

and this will print the array list. Vice versa for numbers consumer interface object as well.

Now if user want to chain both methods (numbers and display) then it can be achieved using default andThen() method of consumer interface.

As –

* **numbers.andThen(display).accept(Arrays.asList(1,2,3,4,5));**

**Note**: System.***out***::println statement in the lambda expression of consumer Interface 2.0 is static method reference. Check Method reference document for it in java 8.

ForEach Consumer Interface Relation

forEach loop in java8 accepts object of consumer interface.

Let’s check below implementation using lambda –

* **Arrays.asList(1,2,3,4,5).forEach(c-> System.out.println(c));**

It is also called internal looping in java8.

Let’s reverse engineer it –

As forEach loop accepts object of consumer interface hence let’s first create object of consumer interface…

As consumer is an interface hence we need to create class that implement Consumer and create it’s object.

So-

Class A implements Consumer{

@override

Public void accept(t){

System.out.println(t);

}

}

Class Main(){

Public static void main(String args[]){

Consumer c = new A();

Arrays.asList(1,2,3,4,5).foreach(c);

}

}

Now, let’s remove us the boiler plate code and rewrite this –

So as consumer is a functional interface hence we can use lambda expression for it and no need to save separate class to implement it as we can define that in anonymous class within the class in which it is been used.

Hence,

Class Main{

Public static void main(String args[]){

// Anonymus class

Consumer<T> c = new Consumer<T>(){

@override

Public void accept(T t){

System.out.println(t);

}

}

Arrays.asList(1,2,3,4,5).foreach(c);

}

}

Let’s re-write in terms of lambda expression-

Class Main{

Public static void main(String args[]){

// Anonymus class

Consumer<T> c = n-> System.out.println(n);

Arrays.asList(1,2,3,4,5).foreach(c);

}

}

So here is what how it is..

continued .. class

System.***out***.println("Traditional way of anonymus class");

// Now explaining consumer interface in terms of Foreach Loop ...

// ForEach method acceps the object of Consumer Interface hence either we can do is like below

Consumer<Integer> c = **new** Consumer<Integer>() {

@Override

**public** **void** accept(Integer t) {

System.***out***.println(t);

}

};

Arrays.*asList*(1,2,3,4,5,6,7).forEach(c);

// Now we know from the short hand property of lambda to write the same in different terms as

Consumer<Integer> c1 = (t)-> System.***out***.println(t);

System.***out***.println("Lambda Expressino....");

// Now this c1 can be passed to foreach method as below

Arrays.*asList*(1,2,3,4,5,6,7).forEach(c1);

System.***out***.println("Method Reference ... ");

// Same thing can also be done usign method reference ...

Arrays.*asList*(1,2,3,4,5,6,7).forEach(System.***out***::println);

}

}