BiConsumer Functional Interface

Before proceeding with this interface I would recommend you to read about [Lambda Calculus](http://data-structure-learning.blogspot.com/2015/05/lambda-calculas.html), [Imperative and Declarative](http://data-structure-learning.blogspot.com/2015/06/java-imperative-and-declarative.html), [Functional Interface](http://data-structure-learning.blogspot.com/2015/06/functional-interfaces-java-8.html) ([with](http://data-structure-learning.blogspot.com/2015/06/functional-interface-with-arguments.html) and [without](http://data-structure-learning.blogspot.com/2015/06/functional-interface-without-arguments.html) arguments). Additional you can read about [Predicate<T> interface](http://data-structure-learning.blogspot.com/p/1.html) (used for filtering based on condition) and Optional<T> class.

BiConsumer interface represents an operation that accepts two arguments and returns no results.

|  |  |
| --- | --- |
| BiConsumer Interface Declaration | BiConsumer interface Declaration with two parameters  **public** **interface** BiConsumer<T, U>  T type of first input argument  U type of second input argument |
| accept() method | **void** accept(T t, U u);  Performs this operation on given arguments |
| andThen() method | **default** BiConsumer<T,U> andThen(BiConsumer<? **super** T, ? **super** U> after) {  Objects.*requireNonNull*(after);  **return** (l, r) -> {  accept(l, r);  after.accept(l, r);  };  }  This method returns the composed BiConsumer that performs this operation followed by after operation. The argument *after* is operated after this operation, the composed version guarantees that. |

Let us write simple BiConsumer that accepts two Integer parameters and print their multiplication.

BiConsumer<Integer, Integer> multiplier = **new** BiConsumer<Integer, Integer>() {

@Override

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

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

}

};

multiplier.accept(10, 20); //Output 200

We just wrote an anonymous inner class but as BiConsumer is Functional Interface we can use lambda operator on it.

Let us refactor the code.

BiConsumer<Integer, Integer> multiplier = (t, u) -> {

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

};

multiplier.accept(10, 20); //Output 200

Now let write 2 different BiConsumer and use andThen() method to execute the composed version of both BiConsumer.

First, let us write multiplier and adder BiConsumer

BiConsumer<Integer, Integer> multiplier = (t, u) -> {

System.***out***.println("Multiplier "+(t \* u));

};

BiConsumer<Integer, Integer> adder = (t, u) -> {

System.***out***.println("Adder "+(t + u));

};

multiplier.andThen(adder).accept(10, 7);//operate multiplier then adder

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

adder.andThen(multiplier).accept(10, 7);//operate adder then multiplier

Output:

Multiplier then adder

Multiplier 70

Adder 17

Adder then multiplier

Adder 17

Multiplier 70

That’s all on BiConsumer interface.