IntPredicate Functional Interface

Previously we have discussed [Predicate Interface](http://data-structure-learning.blogspot.com/p/1.html) and all its methods with examples. I would highly recommend that you read about that first before proceeding to read about IntPredicate. Predicate Interface can accept any type but IntPredicate can only accept input of int primitive type only.

IntPredicate is an int valued predicate which deals with single boolean argument. So IntPredicate is a specialized version (only for int argument) of Predicate interface.

|  |
| --- |
| **IntPredicate Interface Declaration**  **public** **interface** IntPredicate  IntPredicate interface only accepts int argument |
| **test() method**  **boolean** test(**int** value);  test method is used to evaluate this predicate on given argument.  The below code returns true if parameter is greater than 10 else returns false.  IntPredicate intPred = (x) -> x > 10;  System.***out***.println(intPred.test(11)); // Outputs true  System.***out***.println(intPred.test(9)); // Outputs false |
| **and() method**  **default** IntPredicate and(IntPredicate other) {  Objects.*requireNonNull*(other);  **return** (value) -> test(value) && other.test(value);  }  and() returns the composed version of two different IntPredicates. Both the predicates are evaluated in sequence. The composed version represents the short-circuiting logical and of both the predicates. This method will throw NullPointerException is *other* predicate is null.  We will write two different predicates. One checks for number greater than 10 and another check for number less than 20. So it will return true if number is greater than 10 **and** less than 20.  IntPredicate intPredGreater = (x) -> x > 10;  IntPredicate intPredLesser = (x) -> x < 20;  IntPredicate combine = intPredGreater.and(intPredLesser);  System.***out***.println(combine.test(17)); // Outputs true  System.***out***.println(combine.test(25)); // Outputs false |
| **negate() method**  **default** IntPredicate negate() {  **return** (value) -> !test(value);  }  negate() method returns the logical negation of this predicate.  Let us take simple example to demonstrate this method.  We will write a IntPredicate that will return true if number is greater than 10.  IntPredicate intPredGreater = (x) -> x > 10;  System.***out***.println(intPredGreater.test(11)); //Outputs true  System.***out***.println(intPredGreater.test(9)); //Outputs false  Now we will apply negate method on this predicate.  IntPredicate negation=intPredGreater.negate();  System.***out***.println(negation.test(11)); //Outputs false  System.***out***.println(negation.test(9)); //Outputs true  The output is now reversed. |
| **or() method**  **default** IntPredicate or(IntPredicate other) {  Objects.*requireNonNull*(other);  **return** (value) -> test(value) || other.test(value);  }  or() method returns the composed predicate that represents the sort-circuiting logical OR for this predicate and another. While evaluating the composed version of predicate, if this predicate is true then the other predicate is not evaluated. This method will throw NullPointerException is *other* predicate is null.  We will write two different predicates. One check for number greater than 10 and less than 20 and another check for number greater than 40 and less than 50. So it will return true if number is greater than 10 **and** less than 20 **or** number is greater than 40 **and** less than 50.  IntPredicate intPred1 = (x) -> x > 10 && x < 20;  IntPredicate intPred2 = (x) -> x > 40 && x < 50;    **IntPredicate or = intPred1.or(intPred2);**  System.***out***.println(or.test(15)); //Outputs true. Number is between 10 and 20  System.***out***.println(or.test(47)); //Outputs true. Number is between 40 and 50  System.***out***.println(or.test(7)); //Outputs false. Number < 10 and Number < 40  System.***out***.println(or.test(32)); //Outputs false. Number > 20 and Number < 40 |

That’s all on IntPredicate Interface

Read about important java.util.function package’s interface [here](http://data-structure-learning.blogspot.com/p/functional-programming-in-java.html). [Consumer](http://data-structure-learning.blogspot.com/2015/07/java-lambda-consumer-functional.html), [Function](http://data-structure-learning.blogspot.com/2015/07/java-lambda-function-functional.html), [Supplier](http://data-structure-learning.blogspot.com/2015/07/java-lambda-supplier-functional.html), [BinaryOperator](http://data-structure-learning.blogspot.com/2015/07/java-lambda-binaryoperator-functional.html) & [Predicate](http://data-structure-learning.blogspot.com/2015/07/java-lambda-predicate-functional.html) Functional Interfaces. I have also written on [High Order functions](http://data-structure-learning.blogspot.com/2015/07/higher-order-functions-using-function.html) using Function functional interface.