Predicate Functional Interface

Predicate as we know it is a Boolean-valued function in mathematical logic.

Predicate P : X→{true, false} is called predicate on X.

Predicate is true or false depending on the set of values.

P(x) denotes a statement P concerning the variable object x. The set defined by P(x) is written as { x | P(x) }

For example {x | x is even number less than 10} then set is {2, 4, 6, 8}

Now let us talk about the Predicate Interface that is provided in java.util.function package.

Predicate is a [Functional Interface](http://data-structure-learning.blogspot.com/2015/06/functional-interfaces-java-8.html). Read [basics of functional interface](http://data-structure-learning.blogspot.com/2015/06/using-functional-interface.html) [and how to use it with argument](http://data-structure-learning.blogspot.com/2015/06/functional-interface-with-arguments.html) and [without arguments](http://data-structure-learning.blogspot.com/2015/06/functional-interface-without-arguments.html).

We will be using forEach(..) method of Iterable interface, so I recommend you to read the [forEach(..) method post](http://data-structure-learning.blogspot.com/2015/06/iterable-interfaces-foreach-method.html) of Iterable interface.

First we will see what the methods of Predicate Interface are and then we will use them to filter out using conditions.

1. The most important thing for a predicate to do is to evaluate or test.

**boolean** test(T t);

test(T t) method is used to evaluate the predicate on given argument.

1. And operation on this predicate and another. It seems obvious right that we might need two different predicates to filter data. For example get all numbers from 10 to 20(both inclusive) and 30 to 40(both inclusive) but not in between 21 and 29 both inclusive. So we can built predicate like this
   1. {x | x is between 10 and 20}
   2. {y | y is between 30 and 40}
   3. Merge both the predicates we get set {10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40}

Code for and method is as follows:

**default** Predicate<T> and(Predicate<? **super** T> other) {

Objects.*requireNonNull*(other);

**return** (t) -> test(t) && other.test(t);

}

and(..) is the default method that returns the short-circuiting logical AND of this predicate and another.

1. Negate operation. This operation is used to reverse the true to false and false to true. For example we define a Predicate such that x is even. We negate this predicate and we will get output set as odd numbers.

**default** Predicate<T> negate() {

**return** (t) -> !test(t);

}

1. Or operation. Returns a composed predicate that represents a short-circuiting logical or of predicate and another predicate.

**default** Predicate<T> or(Predicate<? **super** T> other) {

Objects.*requireNonNull*(other);

**return** (t) -> test(t) || other.test(t);

}

1. isEqual(Object) operation. It returns the predicate that tests if two arguments are equal according to Object class equals(Object, Object) method.

**static** <T> Predicate<T> isEqual(Object targetRef) {

**return** (**null** == targetRef)

? Objects::*isNull*

: object -> targetRef.equals(object);

}