Predicate Functional Interface

This lesson introduces the Predicate functional interface. We will discuss where a Predicate interface can be used and how to use it.

We'll cover the following

- Introduction to the Predicate interface
  - a) and(Predicate other)
  - b) or(Predicate other)
  - c) negate()
  - d) isEqual(Object targetRef)
- Introduction to the BiPredicate interface

Introduction to the `Predicate` interface

Java 8 provides some in-built functional interfaces in the `java.util.function` package. These interfaces are required so that, while writing lambda expressions, we don't need to worry about creating a functional interface.

There are 43 predefined interfaces in Java 8. Remembering all of them is a bit difficult, so we will divide them into categories and look at each category. The first category that we are looking at in this lesson is `Predicate`.

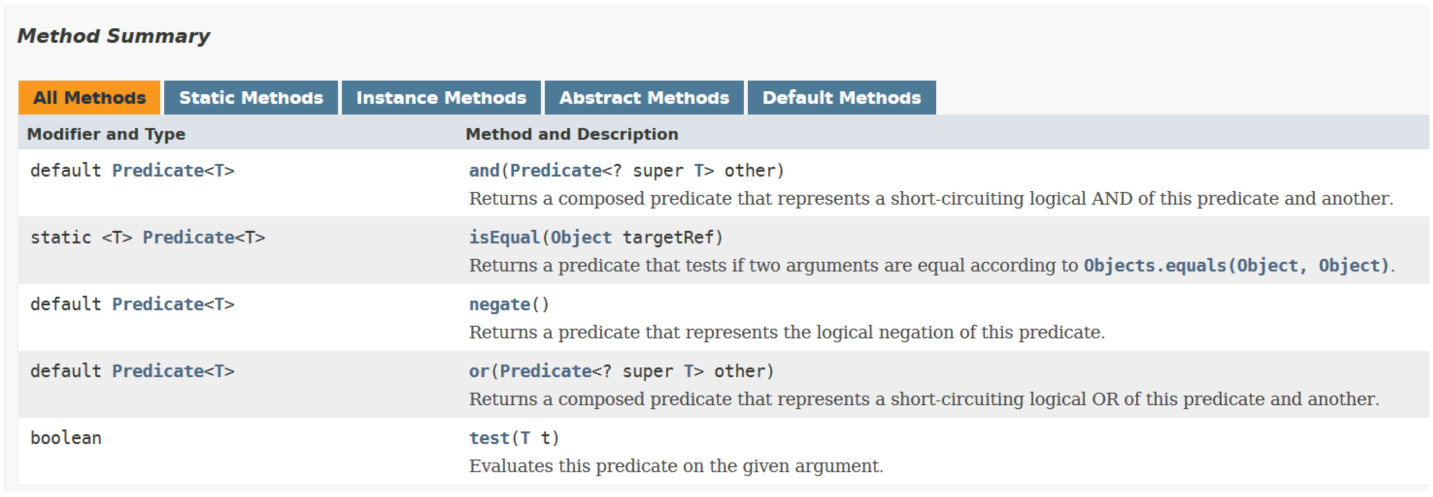
Below are the interfaces in this category:

Interface Name	Description	Abstract Method
`Predicate<T>`	Represents a predicate (boolean-value function) of one argument (reference type)	`boolean test(T t)`

`DoublePredicate`	Accepts one double-value argument	`boolean test(double value)`
`IntPredicate`	Accepts one int-value argument.	`boolean test(int value)`
`LongPredicate`	Accepts one long-value argument	`boolean test(long value)`
`BiPredicate<T,U>`	Accepts two arguments (reference types)	`boolean test(T t, U u)`

The `Predicate<T>` interface has an abstract method `boolean test(T t)`. Basically, a predicate is a function that evaluates the given input and returns true or false.

Below is the list of methods available in `Predicate<T>` interface.


Method Summary

Modifier and Type	Method and Description
`default Predicate<T>`	`and(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
`static <T> Predicate<T>`	`isEqual(Object targetRef)` Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.
`default Predicate<T>`	`negate()` Returns a predicate that represents the logical negation of this predicate.
`default Predicate<T>`	`or(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
`boolean`	`test(T t)` Evaluates this predicate on the given argument.

As we can see, there is only one abstract method and a few default and static methods.

Let's look at an example. We have a `PredicateDemo` class, which has a method `isPersonEligibleForVoting()`. This method takes in a person object and a predicate as a parameter. The predicate is evaluated to check if the given person is eligible for voting or not.

```
import java.util.function.Predicate;

public class PredicateDemo {

  static boolean isPersonEligibleForVoting(Person person, Predicate<Person> predicate){
    return predicate.test(person);
    return predicate.test(person);
  }


  public static void main (String args[]){
    Person person = new Person("Alex", 23);
    // Created a predicate. It returns true if age is greater than 18.
    Predicate<Person> predicate = p -> p.age > 18;

    boolean eligible = isPersonEligibleForVoting(person , predicate);

    System.out.println("Person is eligible for voting: " + eligible);
  }
}

class Person {
  String name;
  int age;

  Person(String name, int age){
    this.name = name;
    this.age = age;
  }
}
```

In the above example, we use a `Predicate<T>`. This interface has some other default and static methods that are used for the purpose of chaining. We will discuss these methods and look at one example for each of them.

a) `and(Predicate other)`

This method returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

In the below example, we need to check if a person is eligible for club membership. The criteria is that the person's age should be more than 18 and less than 60.

We have created two predicates and then combined them into a single predicate using `and()` method.

```
import java.util.function.Predicate;

public class PredicateDemo {

    static boolean isPersonEligibleForMembership(Person person, Predicate<Person> predicate){
        return predicate.test(person);
    }


    public static void main (String args[]){
        Person person = new Person("Alex", 23);

        // Created a predicate. It returns true if age is greater than 18.
        Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
        // Created a predicate. It returns true if age is less than 60.
        Predicate<Person> lessThanSixty = (p) -> p.age < 60;

        Predicate<Person> predicate = greaterThanEighteen.and(lessThanSixty);

        boolean eligible = isPersonEligibleForMembership(person , predicate);
        System.out.println("Person is eligible for membership: " + eligible);
    }
}

class Person {
    String name;
    int age;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

b) `or(Predicate other)`

This method returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

In the below example we need to check if a person is eligible for retirement. The criteria is that either the person's age should be more than 60 or the year of service should be more than 30.

We will create two predicates and then combined them into a single predicate using the `or()` method.

```
import java.util.function.Predicate;

public class PredicateDemo {

    static boolean isPersonEligibleForRetirement(Person person, Predicate<Person> predicate){
        return predicate.test(person);
    }


    public static void main (String args[]){
        Person person = new Person("Alex", 23);
        // Created a predicate. It returns true if age is greater than 18.
        Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
        // Created a predicate. It returns true if year of service is greater than 30.
        Predicate<Person> serviceMoreThanThirty = (p) -> p.yearsOfService > 30;
        Predicate<Person> predicate = greaterThanEighteen.or(serviceMoreThanThirty);

        boolean eligible = isPersonEligibleForRetirement(person , predicate);
        System.out.println("Person is eligible for membership: " + eligible);
    }
}

class Person {
    String name;
    int age;
    int yearsOfService;

    Person(String name, int age){
        this.name = name;
        this.age = age;
        this.yearsOfService = yearsOfService;
    }
}
```

c) `negate()`

This method returns a predicate that represents the logical negation of the predicate it is called on.

Suppose we have a Predicate defined, but in some areas, we need to negate that predicate. In that case, we can use `negate()`.

In the below example, we have a predicate that checks if a number is greater than 10. However, we need to check if a number is less than 10. Now instead of writing a new predicate, we can negate the predicate we already have.

```
import java.util.function.Predicate;

public class PredicateDemo {

    static boolean isNumberLessThanTen(Predicate<Integer> predicate){
        return predicate.negate().test(14);
    }


    public static void main (String args[]){

        Predicate<Integer> numberGreaterThanTen = p -> p > 10;

        boolean isLessThanTen = isNumberLessThanTen( numberGreaterThanTen);
        System.out.println("Is number less than ten: " + isLessThanTen);
    }
}
```

d) `isEqual(Object targetRef)`

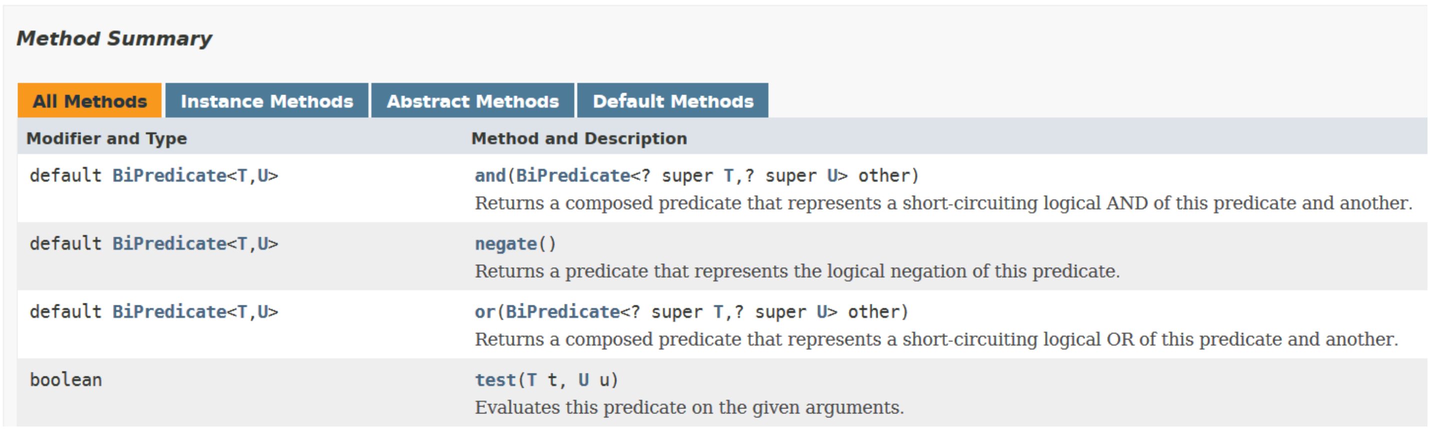
This method returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). This is not a chaining method.

```
import java.util.function.Predicate;

public class PredicateDemo {

  public static void main(String[] args) {
    Predicate<String> predicate  = Predicate.isEqual("Hello");

    // The same thing can be achieved by below lambda.
    // Predicate<String> predicate  = p -> p.equals("Hello");

    System.out.println(predicate.test("Welcome"));
  }
}
```

Introduction to the `BiPredicate` interface

The `Predicate<T>` takes only one parameter and returns the result. Now suppose we have a requirement where we need to send two parameters (i.e person object and min age to vote) and then return the result. Here, we can use `BiPredicate<T, T>`.

The `BiPredicate<T, T>` has a functional method `test(Object, Object)`. It takes in two parameters and returns a boolean value. Below is the list of methods in the `BiPredicate<T, T>` interface.



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`default BiPredicate<T,U>`	`and(BiPredicate<? super T,? super U> other)` Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
`default BiPredicate<T,U>`	`negate()` Returns a predicate that represents the logical negation of this predicate.
`default BiPredicate<T,U>`	`or(BiPredicate<? super T,? super U> other)` Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
`boolean`	`test(T t, U u)` Evaluates this predicate on the given arguments.

If you notice in the above example, we are hard coding the voting age in our lambda, e.g., p -> p.getAge() > 18 . The voting age, i.e., 18, is hardcoded here. If we want to take this age as input, we can use a `BiPredicate` instead of `Predicate`.

In the example shown below, `isPersonEligibleForVoting()` takes in three parameters. Person object, age, and BiPredicate.

```
import java.util.function.BiPredicate;

public class PredicateTest {

  static boolean isPersonEligibleForVoting(
      Person person, Integer minAge, BiPredicate<Person, Integer> predicate) {
    return predicate.test(person, minAge);
  }

  public static void main(String args[]) {
    Person person = new Person("Alex", 23);
    boolean eligible =
        isPersonEligibleForVoting(
            person,
            18,
            (p, minAge) -> {
              return p.age > minAge;
            });
    System.out.println("Person is eligible for voting: " + eligible);
  }
}

class Person {
  String name;
  int age;

  Person(String name, int age){
    this.name = name;
    this.age = age;
  }
}
```

Similarly, we can use other predicates like `IntPredicate`, `LongPredicate`, and `DoublePredicate`. The only difference is that these predicates take an input of a particular type, i.e., int, double, or long.

I hope you now have a clear understanding of using the `Predicate` functional interface in your lambdas.

Here's a brief quiz to check your knowledge!

1 Which functional interface takes in one parameter and returns a boolean?

2

Which of the following is a static method in the `Predicate` interface?

Retake Quiz

In the next lesson, we will look at another category of functional interfaces called the `Supplier` functional interface.