**Introduction**[#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JE9k8XKm8PK#introduction)

Java is a programming language that was developed by James Gosling at Sun Microsystems in 1995. The latest version of java is Java 12, or JDK 12, which was released on March 19th, 2019. Although Java has released its 12th version, Java \* is the most widely used version of Java presently.

Java 8 has introduced a lot of new APIs that have changed the way code is written in Java. Code written in Java 8 is more **concise**, more **readable**, and **faster**, but a developer needs a thorough understanding of the concepts to fully utilize the power of Java 8.

This course aims to help the readers develop a solid understanding of all the features of Java 8 and their practical usages.

**Prerequisites**[#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JE9k8XKm8PK#prerequisites)

The reader must have a basic understanding of the Java language. This course does not teach Java from scratch; it only covers the new features that were introduced in Java 8.

**What you will learn in this course?**[#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JE9k8XKm8PK#what-you-will-learn-in-this-course)

After completing this course, you will be well versed in all the new features introduced in Java 8. Additionally, you will be able to use **lambda expressions** in your code and use the power of **Stream API** to make your code more readable, faster, and more concise.

We will be covering the following topics in this course.

1. Functional interfaces
2. Default and static methods in interfaces
3. Lambda expressions
4. Streams API
5. The new Date and Time API
6. Concurrency enhancements
7. Collections API improvements.

In the next lesson, we will discuss the default methods in interfaces and their purposes.

# Static Methods in interfaces

This lesson explains static methods in interfaces and why they were introduced in Java 8.

**WE'LL COVER THE FOLLOWING**

* + [What are static methods in interfaces?](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gk3AwrqLn8l#what-are-static-methods-in-interfaces)

## What are static methods in interfaces? [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gk3AwrqLn8l#what-are-static-methods-in-interfaces)

The static methods in interfaces are similar to default methods but the only difference is that you can’t override them. Now, why do we need static methods in interfaces if we already have default methods?

Suppose you want to provide some implementation in your interface and you don’t want this implementation to be overridden in the implementing class, then you can declare the method as static.

In the below example, we will defined a Vehicle interface with a static method called cleanVehicle().

1

2

3

4

5

6

7

public interface Vehicle {

    static void cleanVehicle(){

        System.out.println("I am cleaning vehicle");

    }

}





Let us declare a class Car, which implements this Vehicle interface.

Car.java

,

Vehicle.java

,

1

2

3

4

5

6

7

8

9

10

11

12

13

14

public class Car implements Vehicle {

    public void cleanVehicle() {

        System.out.println("Cleaning the vehicle");

    }

    public static void main(String args[]) {

        Car car = new Car();

        car.cleanVehicle();

    }

}





RUN

SAVE \*RESET

###### Output

1.598s

Cleaning the vehicle

In the above interface, we get a compilation error in the Car class because a static method cannot be overridden. Also, since a static method is hidden, we can’t call it from the object of the implementing class. The below code will also not compile.

Car.java

,

Vehicle.java

,

1

2

3

4

5

6

7

8

9

10

public class Car implements Vehicle {

    public static void main(String args[]){

        Car car = new Car();

        car.cleanVehicle();  //This will not compile.

    }

}





RUN

SAVERESET

###### Output

1.201s

Car.java:6: error: cannot find symbol car.cleanVehicle(); //This will not compile. ^ symbol: method cleanVehicle() location: variable car of type Car 1 error

The below class will compile because we are calling the static method that is defined in the interface from the interface reference.

Vehicle.java

,

Car.java

,

1

2

3

4

5

6

7

8

9

public class Car implements Vehicle {

    public static void main(String args[]){

        Car car = new Car();

        Vehicle.cleanVehicle(); //This will compile.

    }

}





RUN

SAVERESET

**Q**

What is a default method?

###### A)

A method that is marked with @Default annotation.

###### B)

A method that cannot be overridden in sub-classes.

**Your Answer**

###### C)

A method that has the implementation inside the interface.

###### D)

None of the above.

#### Great, you got it right!

Retake Quiz

In the next lesson, we will explore functional interfaces

# Functional Interfaces in Java

This lesson explains the concept of functional interfaces which were introduced in Java 8.

**WE'LL COVER THE FOLLOWING**

* + [What are functional interfaces?](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/qZ56WkNA5K7#what-are-functional-interfaces)
  + [What is @FunctionalInterface annotation?](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/qZ56WkNA5K7#what-is-functionalinterface-annotation)

## What are functional interfaces? [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/qZ56WkNA5K7#what-are-functional-interfaces)

An interface that has a single abstract method is called a functional interface.

While an interface can have one or more default methods, it should have only one abstract method to be called a functional interface.

Java 8 has defined the java.util.function package, containing lots of functional interfaces. Some of the functional interfaces defined in Java 8 are Predicate, Consumer, Supplier, Function, etc.

The functional interface is used by lambda expressions. In the next chapter, we will discuss lambdas and also see the usage of some of the functional interfaces in Java 8.

## What is @FunctionalInterface annotation? [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/qZ56WkNA5K7#what-is-functionalinterface-annotation)

Any interface that has only one abstract method can be annotated with the @FunctionalInterface annotation.

This is not mandatory but if an interface is annotated with @FunctionalInterface annotation and someone tries to add another abstract method to the, the compiler will throw an error. Below is an example of a functional interface.

Please add another abstract method in the interface and try to run it. You will see a compilation error.

1

2

3

4

5

6

7

8

9

@FunctionalInterface

public interface Functional {

    void doSomething();

    public static void main(String[] args) {

        System.out.println("foo");

    }

}





RUN

SAVE \*RESET

###### Output

1.625s

foo

If we try to add one more abstract method in the above interface, the compiler shows an error. If an interface is annotated with @FunctionalInterface annotation but does not contain even a single abstract method, then the compiler also complains.

**1**

Is it necessary to declare a functional interface with @FunctionalInterface annotation?

###### A)

Yes

**Your Answer**

###### B)

No

Explanation

Correct! It is not necessary.

**2**

Is it possible to override a static method declared in an interface?

###### A)

Yes

**Your Answer**

###### B)

No

**3**

What would happen if a class inherits two interfaces that, both have default methods with the same name? Select all that apply.

**Selected Option**

###### A)

The compiler will show an error.

###### B)

The code will compile properly

**Not Selected**

###### C)

We will need to override the method in our class.

**SUMMARY**

**Correct2**

**Incorrect1**

#### Great, you got 2 out of 3 correct!

Retake Quiz

That is all about interfaces for now. In the next chapter, we will start exploring lambdas.

# Writing your first lambda

This lesson explains all the basic concepts of lambdas.

**WE'LL COVER THE FOLLOWING**

* + [What is a lambda expression?](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JP2PrpzzjoK#what-is-a-lambda-expression)
  + [How to write a lambda expression](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JP2PrpzzjoK#how-to-write-a-lambda-expression)

## What is a lambda expression? [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JP2PrpzzjoK#what-is-a-lambda-expression)

Java is an object-oriented language. By introducing lambdas in Java 8, the authors of Java tried to add elements of functional programming in Java. Now you might be wondering what the difference between object-oriented programming and functional programming is?

In **object-oriented programming**, objects and classes are the main entities. If we create a function then it should exist within a class. A function has no meaning outside the scope of the class object.

In **functional programming**, functions can exist outside the scope of an object. We can assign them to a reference variable and we can also pass them to other methods as a parameter.

A **lambda expression** is just an anonymous function, i.e., a function with no name and that is not bound to an identifier. We can pass it to other methods as parameters, therefore, using the power of functional programming in Java.

## How to write a lambda expression [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JP2PrpzzjoK#how-to-write-a-lambda-expression)

It might be difficult to understand what lambda is and how to write a lambda without looking at an example. So, let’s look at an example first, and then we will revisit what we just read.

In the below example, we have a functional interface called Greeting. There are two classes HindiGreeting and EnglishGreeting that implement this interface.

1

2

3

4

5

@FunctionalInterface

public interface Greeting {

    void greet();

}





1

2

3

4

5

6

7

8

public class HindiGreeting implements Greeting {

    // Overriding the greet() method from Greeting interface.

    @Override

    public void greet() {

        System.out.println("Namaste");

    }

}





1

2

3

4

5

6

7

8

public class EnglishGreeting implements Greeting {

    // Overriding the greet() method from Greeting interface.

    @Override

    public void greet() {

        System.out.println("Good Morning");

    }

}





Here, we have another class called WellWisher. This class has a method called wish(Greeting greeting) which takes Greeting as a parameter and based on the type of object passed, prints the greeting.

WellWisher.java

,

EnglishGreeting.java

,

HindiGreeting.java

,

Greeting.java

,

1

2

3

4

5

6

7

8

9

10

11

12

13

14

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        Greeting hindiGreeting = new HindiGreeting();

        wish(hindiGreeting);  // Passing an object of HindiGreeting.

        Greeting englishGreeting = new EnglishGreeting();

        wish(englishGreeting);  // Passing an object of EnglishGreeting.

    }

}





RUN

SAVERESET

###### Output

3.344s

Namaste Good Morning

When we run the WellWisher class, we get the below output.

This is a classic object-oriented programming example. Now, what if we want our WellWisher class to greet in all the languages available?

Do we need to create a class for each language, e.g., SpanishGreeting, FrenchGreeting, etc?

Isn’t it possible that we don’t create any class and just send a function to the wish(Greeting greeting) method?

Our wish(Greeting greeting) method will directly execute the function that is provided to it and print the greeting.

This is possible through anonymous classes. We will quickly see how this can be done through an anonymous class, and then jump straight back into lambdas.

In the below example, we will change the WellWisher class to use an anonymous class.

WellWisher.java

,

Greeting.java

,

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        // We are passing an anonymous class object to the wish method.

        wish(new Greeting() {

            @Override

            public void greet() {

                System.out.println("Namaste");

            }

        });

    }

}





RUN

SAVERESET

###### Output

3.134s

Namaste

In the above example, we don’t need to create a class for each language. We can use an anonymous class, and that does the trick for us in the example above. However, don’t you think that this code is still cumbersome? Although the class is anonymous, we are still creating a class.

To make our code less cumbersome, let’s remove all the unnecessary code step-by-step and create our first lambda expression.

**Step 1** -> The compiler knows that the wish(Greeting greeting) method takes in a parameter of type Greeting. So, we don’t need to specifically create an anonymous class of type greeting.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

            public void greet() {

                System.out.println("Namaste");

            }

            );

    }

}





**Step 2** -> We know that the Greeting interface has only one method. So, we don’t need to provide the method name. We are only concerned with the method body.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

            public void () {

                System.out.println("Namaste");

            }

            );

    }

}





**Step 3** -> The compiler can understand that the body does not return anything. So, mentioning the return type is redundant. We can also remove the public declaration.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

                () -> {

                    System.out.println("Namaste");

                }

        );

    }

}





Please note that we add a -> between the empty brackets and the method body. This is how a lambda expression is declared.

There still is one more improvement we can make. Since the method body contains only a single line, the curly braces are also unnecessary.

WellWisher.java

,

Greeting.java

,

1

2

3

4

5

6

7

8

9

10

11

12

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    // Passing a lambda expression to wish method.

    public static void main(String args[]) {

        wish( () -> System.out.println("Namaste") );

    }

}





RUN

SAVERESET

###### Output

2.852s

Namaste

Congratulations!! You have written your first lambda. This is how simple it is to write a lambda expression.

To recap, when we write a lambda expression, we are basically sending a function as a method parameter, and it is directly getting executed.

In the next few lessons, you will see some more examples of writing lambdas. We will also discuss some of the inbuilt functional interfaces and how we can use them to make coding easier.

**1**

Which of the following statements are correct about lambdas?

**Selected Option**

###### A)

They are neither functions nor interfaces.

**Selected Option**

###### B)

Lambda expressions enable functions to be passed as argument.

**Selected Option**

###### C)

Lambdas are denoted with the -> sign.

###### D)

None of the above.

**2**

Which of the following lambdas is incorrect?

**Your Answer**

###### A)

p1,p2 -> p1 + p2

###### B)

(p1,p2) -> p1 + p2

###### C)

(String p1, String p2) -> p1 + p2

###### D)

(p1,p2) -> {return p1 + p2}

**3**

Lambda expressions are compiled to anonymous inner classes?

###### A)

Yes

Explanation

No. Lambdas get wrapped inside new classes generated during runtime.

**Your Answer**

###### B)

No

**SUMMARY**

**Correct2**

**Incorrect1**

#### Great, you got 2 out of 3 correct!

Retake Quiz

In the next lesson, we will look at how to use lambdas to write a comparator.

In this lesson, you will learn how to use the power of lambdas to write a concise comparator.

**WE'LL COVER THE FOLLOWING**

* + [Comparator example using anonymous class](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#comparator-example-using-anonymous-class)
  + [Comparator example using a lambda expression](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#comparator-example-using-a-lambda-expression)

If you’ve been working with Java for some time, then you’ve probably encountered a scenario where you need to sort the elements in a collection.

If your collection contains a wrapper class object then the sorting is very easy. Since all the wrapper classes implement the Comparable interface, you can directly use Collections.sort() to sort your collection.

However, if your collection contains a custom class object then you need to provide the logic to sort your object. In this lesson, we will look at an example in which we will sort a list of Person class objects using a comparator. Then, we will write a program to do the same task using lambdas.

## Comparator example using anonymous class [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#comparator-example-using-anonymous-class)

First, we will create a Person class.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

public class Person {

    private String name;

    private int age;

    private String country;

    public Person(String name, int age, String country) {

        this.name = name;

        this.age = age;

        this.country = country;

    }

    public String getName() {

        return name;

    }

    public int getAge() {

        return age;

    }

    public String getCountry() {

        return country;

    }

}





Now, we have a PersonService class. It has a getPersons(List<Person> persons) method. It takes a list of person objects as input and returns a list of person object in sorted order.

In this method, we are creating an anonymous comparator, which sorts the Person objects on the basis of name.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class PersonService {

    public static List<Person> getPersons(List<Person> persons){

        // Created an anonymous Comparator, which sorts the Person object on the basis of Person name.

        Collections.sort(persons, new Comparator<Person>() {

            @Override

            public int compare(Person p1, Person p2) {

                return p1.getName().compareTo(p2.getName());

            }

        });

        return persons;

    }

}





Finally, we have a PersonMain class that runs our logic.

PersonMain.java

,

PersonService.java

,

Person.java

,

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

import java.util.ArrayList;

import java.util.List;

public class PersonMain {

    public static void main(String args[]){

        List<Person> persons = new ArrayList<>();

        persons.add(new Person("John" , 23 , "USA"));

        persons.add(new Person("Carl" , 23 , "Australia"));

        persons.add(new Person("Amit" , 23 , "India"));

        persons.add(new Person("Vikram" , 23 , "Bhutan"));

        persons.add(new Person("Kane" , 23 , "Brazil"));

        // Calling getPerson() method which will return the List of Person in sorted order.

        List<Person> sortedPersons = PersonService.getPersons(persons);

        System.out.println("Persons after sorting");

        // Printing the name of each person.

        for(Person person : sortedPersons){

            System.out.println("Person Name : " + person.getName());

        }

    }

}





RUN

SAVERESET

If you look at the Comparator interface, you notice that it is a functional interface. It has only one abstract method called compare(). This makes it a perfect candidate to be used in lambdas.

## Comparator example using a lambda expression [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#comparator-example-using-a-lambda-expression)

Now, let’s see how we can write the same logic using a lambda expression. As discussed in the previous lesson, when writing lambdas, we only need to consider the input parameters and the method body.

Below is the signature of the compare() method.

int compare(T o1, T o2)

It takes two parameters as input and returns an int.

Let’s start constructing the lambda expression:

The structure of lambda will be like:

(p1, p2) -> {};

Here, p1 and p2 are the two input parameters. We can name them anything.

Now, we will add the body.

(p1, p2) -> p1.getName().compareTo(p2.getName());

So, this is the lambda expression for sorting the Person objects based on age.

You can see how easy and concise it is to write code with lambdas instead of using anonymous classes.

PersonService.java

,

PersonMain.java

,

Person.java

,

1

2

3

4

5

6

7

8

9

10

11

12

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class PersonService {

    public static List<Person> getPersons(List<Person> persons) {

        // Instead of creating an anonymous class, we have provided a lambda expression.

        Collections.sort(persons, (p1, p2) -> p1.getName().compareTo(p2.getName()));

        return persons;

    }

}



RUN

SAVERESET

In the next lesson, you will learn about the Predicate functional interface.

# 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](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#introduction-to-the-predicate-interface)
    - [a) and(Predicate other)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#a-andpredicate-other)
    - [b) or(Predicate other)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#b-orpredicate-other)
    - [c) negate()](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#c-negate)
    - [d) isEqual(Object targetRef)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#d-isequalobject-targetref)
  + [Introduction to the BiPredicate interface](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#introduction-to-the-bipredicate-interface)

## Introduction to the Predicate interface [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#introduction-to-the-predicate-interface)     - [a) and(Predicate other)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#a-andpredicate-other)     - [b) or(Predicate other)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#b-orpredicate-other)     - [c) negate()](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#c-negate)     - [d) isEqual(Object targetRef)](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#d-isequalobject-targetref)   + [Introduction to the BiPredicate interface](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#introduction-to-the-bipredicate-interface)  Introduction to the Predicate interface [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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.  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.  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  import java.util.function.Predicate;    public class PredicateDemo {      static boolean isPersonEligibleForVoting(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> 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;    }  }      RUN  SAVERESET Output 4.498s  Person is eligible for voting: true  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) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#a-andpredicate-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.  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  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;      }  }      RUN  SAVERESET Output 4.361s  Person is eligible for membership: true b) or(Predicate other) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#b-orpredicate-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.  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  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;      }  }      RUN  SAVERESET Output 7.444s  Person is eligible for membership: true c) negate() [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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.  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  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);      }  }      RUN  SAVERESET d) isEqual(Object targetRef) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#d-isequalobject-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.  1  2  3  4  5  6  7  8  9  10  11  12  13  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"));    }  }      RUN  SAVERESET Introduction to the BiPredicate interface [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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.  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.  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  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;    }  }      RUN  SAVERESET Output 2.284s  Person is eligible for voting: true  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? A) BiPredicate  **Your Answer** B) Predicate C) Function D) None of the above.  **2**  Which of the following is a static method in the Predicate interface? A) or() B) and()  **Your Answer** C) negate()  **Correct Answer** D) isEqual()  **SUMMARY**  **Correct1**  **Incorrect1**  Retake Quiz  In the next lesson, we will look at another category of functional interfaces called the Supplier functional interface.  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.

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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

import java.util.function.Predicate;

public class PredicateDemo {

  static boolean isPersonEligibleForVoting(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> 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;

  }

}





RUN

SAVERESET

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) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#a-andpredicate-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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

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;





RUN

SAVERESET

### b) or(Predicate other) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#b-orpredicate-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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

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;





RUN

SAVERESET

### c) negate() [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

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);

    }

}





RUN

SAVERESET

### d) isEqual(Object targetRef) [**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#d-isequalobject-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.

1

2

3

4

5

6

7

8

9

10

11

12

13

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"));

  }

}





RUN

SAVERESET

## Introduction to the BiPredicate interface [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/mEKVvqrRwqO#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.

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.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

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;

  }

}





RUN

SAVERESET

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?

###### A)

BiPredicate

**Your Answer**

###### B)

Predicate

###### C)

Function

###### D)

None of the above.

**2**

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

###### A)

or()

###### B)

and()

**Your Answer**

###### C)

negate()

**Correct Answer**

###### D)

isEqual()

**SUMMARY**

**Correct1**

**Incorrect1**

Retake Quiz

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

**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj)

# Supplier Functional Interface

In this lesson, we will look at the supplier functional interface.

**WE'LL COVER THE FOLLOWING**

* + [Supplier<T>](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#suppliert)
  + [IntSupplier](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#intsupplier)
  + [DoubleSupplier](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#doublesupplier)

Supplier is an interface that does not take in any argument but produces a value when the get() function is invoked. Suppliers are useful when we don’t need to supply any value and obtain a result at the same time.

Below are some of the functional interfaces, which can be categorized as a supplier.

| Interface Name | Description | Abstract Method |
| --- | --- | --- |
| Supplier<T> | Represents a supplier of results (reference type) | T get() |
| DoubleSupplier | A supplier of double-value results | double getAsDouble() |
| IntSupplier | A supplier of int-value results | int getAsInt() |
| LongSupplier | A supplier of long-value results | long getAsLong() |
| BooleanSupplier | A supplier of boolean-value results | boolean getAsBoolean() |

## Supplier<T> [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#suppliert)

The Supplier<T> interface supplies a result of type T. In the previous lesson, we were passing a person object and a predicate to our isPersonEligibleForVoting() method.

In this example, we will provide a Supplier<Person> instead of the Person object. The isPersonEligibleForVoting() method will, itself, fetch the Person object from the supplier. Here is the code for this.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

import java.util.function.Predicate;

import java.util.function.Supplier;

public class SupplierTest {

  static boolean isPersonEligibleForVoting(

      Supplier<Person> supplier, Predicate<Person> predicate) {

    return predicate.test(supplier.get());

  }

  public static void main(String args[]) {

    Supplier<Person> supplier = () -> new Person("Alex", 23);

    Predicate<Person> predicate = (p) -> p.age > 18;

    boolean eligible =

        isPersonEligibleForVoting(supplier, 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;

  }

}





RUN

SAVERESET

The Supplier<T> interface does not contain any default or static methods. Let us look at some of the primitive specializations of the supplier interface.

## IntSupplier [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#intsupplier)

The IntSupplier interface has a method getAsInt(), which applies the given operation on its argument and returns an int value. It is similar to using an object of type Supplier<Integer>.

1

2

3

4

5

6

7

8

9

10

11

12

import java.util.function.IntSupplier;

public class SupplierDemo {

  public static void main(String args[]) {

        IntSupplier supplier = () -> (int)(Math.random() \* 10);

        System.out.println(supplier.getAsInt());

  }

}





RUN

SAVERESET

## DoubleSupplier [#](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/gkAAgzM6V2j#doublesupplier)

The DoubleSupplier interface has a method getAsDouble(), which applies the given operation on its argument and returns a double value. It is similar to using an object of type Supplier<Double>.

1

2

3

4

5

6

7

8

9

10

11

12

import java.util.function.DoubleSupplier;

public class SupplierDemo {

  public static void main(String args[]) {

        DoubleSupplier supplier = () -> (int)(Math.random() \* 10);

        System.out.println(supplier.getAsDouble());

  }

}





RUN

SAVERESET

**1**

What does T represent in Supplier<T> ?

###### A)

The type of input parameter

**Your Answer**

###### B)

The return type

###### C)

None of the above

**2**

What is the default method of IntSupplier?

###### A)

get()

###### B)

getInt()

**Your Answer**

###### C)

getAsInt()

###### D)

None of the above.

**SUMMARY**

**Correct2**

**Incorrect0**

#### Awesome! you got all 2 correct!

Retake Quiz

In the next lesson, we will look at the Consumer functional interfaces. These interfaces are the opposite of suppliers.

