<https://www.baeldung.com/java-8-functional-interfaces>

**1. Introduction**

This tutorial is a guide to different functional interfaces present in Java 8, as well as their general use cases, and usage in the standard JDK library.

## ****2. Lambdas in Java 8****

Java 8 brought a powerful new syntactic improvement in the form of lambda expressions. **A lambda is an anonymous function that we can handle as a first-class language citizen. For instance, we can pass it to or return it from a method**.

Before Java 8, we would usually create a class for every case where we needed to encapsulate a single piece of functionality. This implied a lot of unnecessary boilerplate code to define something that served as a primitive function representation.

The article [“Lambda Expressions and Functional Interfaces: Tips and Best Practices”](https://www.baeldung.com/java-8-lambda-expressions-tips) describes in more detail the functional interfaces and best practices of working with lambdas. **This guide focuses on some particular functional interfaces that are present in the java.util.function package**.

## ****3. Functional Interfaces****

**It's recommended that all functional interfaces have an informative @FunctionalInterface annotation**. This clearly communicates the purpose of the interface, and also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

**Any interface with a SAM(Single Abstract Method) is a functional interface**, **and its implementation may be treated as lambda expressions.**

Note that Java 8's default methods are not abstract and do not count; a functional interface may still have multiple default methods. We can observe this by looking at the Function's [documentation](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html).

<https://dzone.com/articles/functional-interface-explained-in-detail-introduce>

Functional interfaces are introduced as part of Java 8.

It is implemented using the annotation called **@FunctionalInterface**.

It ensures that the interface should have only one abstract method.

The usage of the abstract keyword is optional as the method defined inside interface is by default abstract.

It is important to note that a functional interface can have multiple default methods (it can be said concrete methods which are default) but only one abstract method.

The default method has been introduced in interface so that a new method can be appended in the class without affecting the implementing class of the existing interfaces. Because prior to java 8 the implementing class of an interface has to implement all the abstract methods defined in the interface.

The functional interface has been introduced in Java 8 to support the lambda expression in java 8 on the other hand **it can be said lambda expression is the instance of functional interface**.

For example:

In Java 8 there are 4 main functional interfaces are introduced which could be used in different scenarios. These are given below.

1. Consumer
2. Predicate
3. Function
4. Supplier

Among the above four interfaces, the first three interfaces also have extensions also which are given below.

Consumer - BiConsumer

Predicate – BiPredicate

Function – BiFunction, UnaryOperator, BinaryOperator

1. CONSUMER

Let’s discuss about Consumer.

* The consumer interface accepts one argument but there is no return value.
* The name of function inside the CONSUMER interface is **accept**.
* BICONSUMER

The extension of the Consumer which is BiConsumer accepts two arguments and returns nothing.

1. PREDICATE

Predicate will accept one argument, do some processing and then return boolean.

* The name of function inside the PREDICATE interface is **test**.
* BIPREDICATE

Instead of one argument BiPredicate will accept two arguments and return nothing.

3. FUNCTION

This interface accepts one argument and returns a value after the required processing. It is defined as below. The required processing logic will be executed on invocation of the apply method.

* The name of function inside the FUNCTION interface is **apply**.
* BIFUNCTION

The BiFunction is similar to Function except it accepts two inputs whereas Function accepts one argument.

* UNARYOPERATOR and BINARYOPERATOR

Another two interfaces are UnaryOperator and BinaryOperator which extends the Function and BiFunction respectively.

The UnaryOperator accepts a single argument and return a single argument but both the input and output argument should be of same of similar type.

On the other hand BinaryOperator accepts two arguments and return one argument similar to BiFunction but type of all the input and output argument should be of similar type.

4. SUPPLIER

Supplier functional interface does not accept any input rather return single output.

* The name of function inside the SUPPLIER interface is **get**.

<https://www.infoworld.com/article/3452018/get-started-with-lambda-expressions.html>

# Get started with lambda expressions in Java

### Learn how to use lambda expressions and functional programming techniques in your Java programs

Before Java SE 8, anonymous classes were typically used to pass functionality to a method. This practice obfuscated source code, making it harder to understand. Java 8 eliminated this problem by introducing lambdas. This tutorial first introduces the lambda language feature, then provides a more detailed introduction to functional programming with lambda expressions along with target types. You'll also learn how lambdas interact with scopes, local variables, the this and super keywords, and Java exceptions.

Note that code examples in this tutorial are compatible with JDK 12.

## Lambdas: A primer

A lambda expression (lambda) describes a block of code (an [anonymous function](http://en.wikipedia.org/wiki/Anonymous_function)) that can be passed to constructors or methods for subsequent execution. The constructor or method receives the lambda as an argument. Consider the following example:

() -> **System**.**out**.println("Hello")

This example identifies a lambda for outputting a message to the standard output stream. From left to right, **() identifies the lambda's formal parameter list** (there are no parameters in the example),

**-> indicates that the expression is a lambda**, and System.out.println("Hello") is the code to be executed.

Lambdas simplify the use of functional interfaces, which are annotated interfaces that each declares exactly one abstract method (although they can also declare any combination of default, static, and private methods). For example, the standard class library provides a java.lang.Runnable interface with a single abstract void run() method. This functional interface's declaration appears below:

@FunctionalInterface

**public** **interface** **Runnable**

{

**public** **abstract** **void** run();

}

The class library annotates Runnable with @FunctionalInterface, which is an instance of the java.lang.FunctionalInterface annotation type. FunctionalInterface is used to annotate those interfaces that are to be used in lambda contexts.

A lambda doesn't have an explicit interface type. Instead, the compiler uses the surrounding context to infer which functional interface to instantiate when a lambda is specified--the lambda is bound to that interface. For example, suppose I specified the following code fragment, which passes the previous lambda as an argument to the java.lang.Thread class's Thread(Runnable target) constructor:

**new** **Thread**(() -> **System**.**out**.println("Hello"));

The compiler determines that the lambda is being passed to Thread(Runnable r) because this is the only constructor that satisfies the lambda: Runnable is a functional interface, the lambda's empty formal parameter list () matches run()'s empty parameter list, and the return types (void) also agree. The lambda is bound to Runnable.

Listing 1 presents the source code to a small application that lets you play with this example.

#### Listing 1. LambdaDemo.java (version 1)

**public** **class** **LambdaDemo**

{

**public** **static** **void** main(**String**[] args)

{

**new** **Thread**(() -> **System**.**out**.println("Hello")).start();

}

}

Compile Listing 1 (javac LambdaDemo.java) and run the application (java LambdaDemo). You should observe the following output:

Hello

Lambdas can greatly simplify the amount of source code that you must write, and can also make source code much easier to understand. For example, without lambdas, you would probably specify Listing 2's more verbose code, which is based on an instance of an anonymous class that implements Runnable.

#### Listing 2. LambdaDemo.java (version 2)

**public** **class** **LambdaDemo**

{

**public** **static** **void** main(**String**[] args)

{

**Runnable** r = **new** **Runnable**()

{

@Override

**public** **void** run()

{

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

}

};

**new** **Thread**(r).start();

}

}

After compiling this source code, run the application. You'll discover the same output as previously shown.

## Java lambdas in depth

To use lambdas effectively, you must understand the syntax of lambda expressions along with the notion of a target type. You also need to understand how lambdas interact with scopes, local variables, the this and super keywords, and exceptions. I'll cover all of these topics in the sections that follow.

### Lambda syntax

Every lambda conforms to the following syntax:

( formal-parameter-list ) -> { expression-or-statements }

The *formal-parameter-list* is a comma-separated list of formal parameters, which must match the parameters of a functional interface's single abstract method at runtime. If you omit their types, the compiler infers these types from the context in which the lambda is used. Consider the following examples:

(**double** a, **double** b) *// types explicitly specified*

(a, b) *// types inferred by compiler*

You must specify parentheses for multiple or no formal parameters. However, you can omit the parentheses (although you don't have to) when specifying a single formal parameter. (This applies to the parameter name only--parentheses are required when the type is also specified.) Consider the following additional examples:

x *// parentheses omitted due to single formal parameter*

(**double** x) *// parentheses required because type is also present*

() *// parentheses required when no formal parameters*

(x, y) *// parentheses required because of multiple formal parameters*

The *formal-parameter-list* is followed by a -> token, which is followed by *expression-or-statements*--an expression or a block of statements (either is known as the lambda's body). Unlike expression-based bodies, statement-based bodies must be placed between open ({) and close (}) brace characters:

(**double** radius) -> **Math**.PI \* radius \* radius

radius -> { **return** **Math**.PI \* radius \* radius; }

radius -> { **System**.**out**.println(radius); **return** **Math**.PI \* radius \* radius; }

The first example's expression-based lambda body doesn't have to be placed between braces. The second example converts the expression-based body to a statement-based body, in which return must be specified to return the expression's value. The final example demonstrates multiple statements and cannot be expressed without the braces.

Listing 3 presents a simple application that demonstrates lambda syntax; note that this listing builds on the previous two code examples.

#### Listing 3. LambdaDemo.java (version 3)

@FunctionalInterface

**interface** **BinaryCalculator**

{

**double** calculate(**double** value1, **double** value2);

}

@FunctionalInterface

**interface** **UnaryCalculator**

{

**double** calculate(**double** value);

}

**public** **class** **LambdaDemo**

{

**public** **static** **void** main(**String**[] args)

{

**System**.**out**.printf("18 + 36.5 = %f%n", calculate((**double** v1, **double** v2) ->

v1 + v2, 18, 36.5));

**System**.**out**.printf("89 / 2.9 = %f%n", calculate((v1, v2) -> v1 / v2, 89,

2.9));

**System**.**out**.printf("-89 = %f%n", calculate(v -> -v, 89));

**System**.**out**.printf("18 \* 18 = %f%n", calculate((**double** v) -> v \* v, 18));

}

**static** **double** calculate(**BinaryCalculator** calc, **double** v1, **double** v2)

{

**return** calc.calculate(v1, v2);

}

**static** **double** calculate(**UnaryCalculator** calc, **double** v)

{

**return** calc.calculate(v);

}

}

Listing 3 first introduces the BinaryCalculator and UnaryCalculator functional interfaces whose calculate() methods perform calculations on two input arguments or on a single input argument, respectively. This listing also introduces a LambdaDemo class whose main() method demonstrates these functional interfaces.

The functional interfaces are demonstrated in the static double calculate(BinaryCalculator calc, double v1, double v2) and static double calculate(UnaryCalculator calc, double v) methods. The lambdas pass code as data to these methods, which are received as BinaryCalculator or UnaryCalculator instances.

Compile Listing 3 and run the application. You should observe the following output:

18 + 36.5 = 54.500000

89 / 2.9 = 30.689655

-89 = -89.000000

18 \* 18 = 324.000000

### Target types

A lambda is associated with an implicit target type, which identifies the type of object to which a lambda is bound. The target type must be a functional interface that's inferred from the context, which limits lambdas to appearing in the following contexts:

* Variable declaration
* Assignment
* Return statement
* Array initializer
* Method or constructor arguments
* Lambda body
* Ternary conditional expression
* Cast expression

Listing 4 presents an application that demonstrates these target type contexts.

#### Listing 4. LambdaDemo.java (version 4)

**import** java.io.**File**;

**import** java.io.**FileFilter**;

**import** java.nio.file.**Files**;

**import** java.nio.file.**FileSystem**;

**import** java.nio.file.**FileSystems**;

**import** java.nio.file.**FileVisitor**;

**import** java.nio.file.**FileVisitResult**;

**import** java.nio.file.**Path**;

**import** java.nio.file.**PathMatcher**;

**import** java.nio.file.**Paths**;

**import** java.nio.file.**SimpleFileVisitor**;

**import** java.nio.file.attribute.**BasicFileAttributes**;

**import** java.security.**AccessController**;

**import** java.security.**PrivilegedAction**;

**import** java.util.**Arrays**;

**import** java.util.**Collections**;

**import** java.util.**Comparator**;

**import** java.util.**List**;

**import** java.util.concurrent.**Callable**;

**public** **class** **LambdaDemo**

{

**public** **static** **void** main(**String**[] args) **throws** **Exception**

{

*// Target type #1: variable declaration*

**Runnable** r = () -> { **System**.**out**.println("running"); };

r.run();

*// Target type #2: assignment*

r = () -> **System**.**out**.println("running");

r.run();

*// Target type #3: return statement (in getFilter())*

**File**[] files = **new** **File**(".").listFiles(getFilter("txt"));

**for** (**int** i = 0; i < files.length; i++)

**System**.**out**.println(files[i]);

*// Target type #4: array initializer*

**FileSystem** fs = **FileSystems**.getDefault();

**final** **PathMatcher** matchers[] =

{

(path) -> path.toString().endsWith("txt"),

(path) -> path.toString().endsWith("java")

};

**FileVisitor**<**Path**> visitor;

visitor = **new** **SimpleFileVisitor**<**Path**>()

{

@Override

**public** **FileVisitResult** visitFile(**Path** file,

**BasicFileAttributes** attribs)

{

**Path** name = file.getFileName();

**for** (**int** i = 0; i < matchers.length; i++)

{

**if** (matchers[i].matches(name))

**System**.**out**.printf("Found matched file: '%s'.%n",

file);

}

**return** **FileVisitResult**.CONTINUE;

}

};

**Files**.walkFileTree(**Paths**.**get**("."), visitor);

*// Target type #5: method or constructor arguments*

**new** **Thread**(() -> **System**.**out**.println("running")).start();

*// Target type #6: lambda body (a nested lambda)*

**Callable**<**Runnable**> callable = () -> () ->

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

callable.call().run();

*// Target type #7: ternary conditional expression*

**boolean** ascendingSort = **false**;

**Comparator**<**String**> cmp;

cmp = (ascendingSort) ? (s1, s2) -> s1.compareTo(s2)

: (s1, s2) -> s2.compareTo(s1);

**List**<**String**> cities = **Arrays**.asList("Washington", "London", "Rome",

"Berlin", "Jerusalem", "Ottawa",

"Sydney", "Moscow");

**Collections**.sort(cities, cmp);

**for** (**int** i = 0; i < cities.size(); i++)

**System**.**out**.println(cities.**get**(i));

*// Target type #8: cast expression*

**String** user = **AccessController**.doPrivileged((**PrivilegedAction**<**String**>) ()

-> **System**.getProperty("user.name"));

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

}

**static** **FileFilter** getFilter(**String** ext)

{

**return** (pathname) -> pathname.toString().endsWith(ext);

}

}