**Java 8 Functional Interfaces**

JULY 11, 2016 BY [PANKAJ](http://www.journaldev.com/author/pankaj) [13 COMMENTS](http://www.journaldev.com/2763/java-8-functional-interfaces#comments)

Welcome to the Java 8 functional interfaces example tutorial. Java has always been an **Object Oriented Programming** language. What is means that everything in java programming revolves around Objects (except some primitive types for simplicity). We don’t have only functions in java, they are part of Class and we need to use the class/object to invoke any function.

**Java 8 Functional Interfaces**

If we look into some other programming languages such as C++, JavaScript; they are called **functional programming language** because we can write functions and use them when required. Some of these languages support Object Oriented Programming as well as Functional Programming.

Being object oriented is not bad, but it brings a lot of verbosity to the program. For example, let’s say we have to create an instance of Runnable. Usually we do it using anonymous classes like below.

Runnable r = new Runnable(){

@Override

public void run() {

System.out.println("My Runnable");

}};

If you look at the above code, the actual part that is of use is the code inside run() method. Rest all of the code is because of the way java programs are structured.

Java 8 Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code by removing a lot of boiler-plate code.

**Java 8 Functional Interface**

An interface with exactly one abstract method is called Functional Interface. @FunctionalInterface annotation is added so that we can mark an interface as functional interface.

It is not mandatory to use it, but it’s best practice to use it with functional interfaces to avoid addition of extra methods accidentally. If the interface is annotated with @FunctionalInterface annotation and we try to have more than one abstract method, it throws compiler error.

The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

Java 8 Collections API has been rewritten and new Stream API is introduced that uses a lot of functional interfaces. Java 8 has defined a lot of functional interfaces in java.util.function package. Some of the useful java 8 functional interfaces are Consumer, Supplier, Function and Predicate.

You can find more detail about them in [Java 8 Stream Example](http://www.journaldev.com/2774/java-8-stream).

java.lang.Runnable is a great example of functional interface with single abstract method run().

Below code snippet provides some guidance for functional interfaces:

interface Foo { boolean equals(Object obj); }

// Not functional because equals is already an implicit member (Object class)

interface Comparator<T> {

boolean equals(Object obj);

int compare(T o1, T o2);

}

// Functional because Comparator has only one abstract non-Object method

interface Foo {

int m();

Object clone();

}

// Not functional because method Object.clone is not public

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<String> arg); }

interface Z extends X, Y {}

// Functional: two methods, but they have the same signature

interface X { Iterable m(Iterable<String> arg); }

interface Y { Iterable<String> m(Iterable arg); }

interface Z extends X, Y {}

// Functional: Y.m is a subsignature & return-type-substitutable

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<Integer> arg); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { int m(Iterable<String> arg, Class c); }

interface Y { int m(Iterable arg, Class<?> c); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { long m(); }

interface Y { int m(); }

interface Z extends X, Y {}

// Compiler error: no method is return type substitutable

interface Foo<T> { void m(T arg); }

interface Bar<T> { void m(T arg); }

interface FooBar<X, Y> extends Foo<X>, Bar<Y> {}

// Compiler error: different signatures, same erasure

**Lambda Expression**

Lambda Expression are the way through which we can visualize **functional programming** in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that’s why Java language provide support for using lambda expressions only with functional interfaces.

Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is **(argument) -> (body)**. Now let’s see how we can write above anonymous Runnable using lambda expression.

Runnable r1 = () -> System.out.println("My Runnable");

Let’s try to understand what is happening in the lambda expression above.

* Runnable is a functional interface, that’s why we can use lambda expression to create it’s instance.
* Since run() method takes no argument, our lambda expression also have no argument.
* Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

**Why do we need Lambda Expression**

1. **Reduced Lines of Code**  
   One of the clear benefit of using lambda expression is that the amount of code is reduced, we have already seen that how easily we can create instance of a functional interface using lambda expression rather than using anonymous class.
2. **Sequential and Parallel Execution Support**

Another benefit of using lambda expression is that we can benefit from the Stream API sequential and parallel operations support.

To explain this, let’s take a simple example where we need to write a method to test if a number passed is prime number or not.

Traditionally we would write it’s code like below. The code is not fully optimized but good for example purpose, so bear with me on this.

//Traditional approach

private static boolean isPrime(int number) {

if(number < 2) return false;

for(int i=2; i<number; i++){

if(number % i == 0) return false;

}

return true;

}

The problem with above code is that it's sequential in nature, if the number is very huge then it will take significant amount of time. Another problem with code is that there are so many exit points and it's not readable. Let's see how we can write the same method using lambda expressions and stream API.

//Declarative approach

private static boolean isPrime(int number) {

return number > 1

&& IntStream.range(2, number).noneMatch(

index -> number % index == 0);

}

IntStream is a sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.

For more readability, we can also write the method like below.

private static boolean isPrime(int number) {

IntPredicate isDivisible = index -> number % index == 0;

return number > 1

&& IntStream.range(2, number).noneMatch(

isDivisible);

}

If you are not familiar with IntStream, it's range() method returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

noneMatch() method returns whether no elements of this stream match the provided predicate. It may not evaluate the predicate on all elements if not necessary for determining the result.

1. **Passing Behaviors into methods**

Let's see how we can use lambda expressions to pass behavior of a method with a simple example. Let's say we have to write a method to sum the numbers in a list if they match a given criteria. We can use Predicate and write a method like below.

public static int sumWithCondition(List<Integer> numbers, Predicate<Integer> predicate) {

return numbers.parallelStream()

.filter(predicate)

.mapToInt(i -> i)

.sum();

}

Sample usage:

//sum of all numbers

sumWithCondition(numbers, n -> true)

//sum of all even numbers

sumWithCondition(numbers, i -> i%2==0)

//sum of all numbers greater than 5

sumWithCondition(numbers, i -> i>5)

1. **Higher Efficiency with Laziness**

One more advantage of using lambda expression is the lazy evaluation, for example let's say we need to write a method to find out the maximum odd number in the range 3 to 11 and return square of it.

Usually we will write code for this method like this:

private static int findSquareOfMaxOdd(List<Integer> numbers) {

int max = 0;

for (int i : numbers) {

if (i % 2 != 0 && i > 3 && i < 11 && i > max) {

max = i;

}

}

return max \* max;

}

Above program will always run in sequential order but we can use Stream API to achieve this and get benefit of Laziness-seeking. Let's see how we can rewrite this code in functional programming way using Stream API and lambda expressions.

public static int findSquareOfMaxOdd(List<Integer> numbers) {

return numbers.stream()

.filter(NumberTest::isOdd) //Predicate is functional interface and

.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it

.filter(NumberTest::isLessThan11) // rather than anonymous inner classes

.max(Comparator.naturalOrder())

.map(i -> i \* i)

.get();

}

public static boolean isOdd(int i) {

return i % 2 != 0;

}

public static boolean isGreaterThan3(int i){

return i > 3;

}

public static boolean isLessThan11(int i){

return i < 11;

}

If you are surprised with the double colon (::) operator, it's introduced in Java 8 and used for **method references**. Java Compiler takes care of mapping the arguments to the called method. It's short form of lambda expressions i -> isGreaterThan3(i) or i -> NumberTest.isGreaterThan3(i).

**Lambda Expression Examples**

Below I am providing some code snippets for lambda expressions with small comments explaining them.

() -> {} // No parameters; void result

() -> 42 // No parameters, expression body

() -> null // No parameters, expression body

() -> { return 42; } // No parameters, block body with return

() -> { System.gc(); } // No parameters, void block body

// Complex block body with multiple returns

() -> {

if (true) return 10;

else {

int result = 15;

for (int i = 1; i < 10; i++)

result \*= i;

return result;

}

}

(int x) -> x+1 // Single declared-type argument

(int x) -> { return x+1; } // same as above

(x) -> x+1 // Single inferred-type argument, same as below

x -> x+1 // Parenthesis optional for single inferred-type case

(String s) -> s.length() // Single declared-type argument

(Thread t) -> { t.start(); } // Single declared-type argument

s -> s.length() // Single inferred-type argument

t -> { t.start(); } // Single inferred-type argument

(int x, int y) -> x+y // Multiple declared-type parameters

(x,y) -> x+y // Multiple inferred-type parameters

(x, final y) -> x+y // Illegal: can't modify inferred-type parameters

(x, int y) -> x+y // Illegal: can't mix inferred and declared types

**Method and Constructor References**

A method reference is used to refer to a method without invoking it; a constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.

Examples of method and constructor references:

System::getProperty

System.out::println

"abc"::length

ArrayList::new

int[]::new

That's all for Java 8 Functional Interfaces and Lambda Expression Tutorial. I would strongly suggest to look into using it because this syntax is new to Java and it will take some time to grasp it.

You should also check out [Java 8 Features](http://www.journaldev.com/2389/java-8-features-with-examples) to learn about all the improvements and changes in Java 8 release.

FILED UNDER: [JAVA](http://www.journaldev.com/dev/java)

**About Pankaj**

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on [**Google Plus**](https://plus.google.com/118104420597648001532/posts?rel=author), **[Facebook](https://www.facebook.com/journaldev)** or [**Twitter**](https://twitter.com/JournalDev). I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **[Youtube](https://www.youtube.com/user/JournalDev)**.

[« Java 8 Interface Changes – static method, default method](http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method)

[Java 8 Stream – Java Stream »](http://www.journaldev.com/2774/java-8-stream)

**Comments**

1. **sawai says**

[NOVEMBER 21, 2016 AT 10:15 PM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36888)

I am new in Functional Interface and today i am learning from few of tutorial sites. I have a question plz provide your suggestions and guide me.

Below mentioned code have an question for me.

@FunctionalInterface  
interface Demo {  
Object clone(); // protected  
//int hashCode(); // public  
//boolean equals(Object c); // public  
//public void wait(); // final so we cannot override this one.  
}

> Object class is parent for all java classes.  
> here wait() method says not overrided bcoz this one is final. So its mean Demo interface also child of Object class (in general terms).

> @FunctionalInterface means interface with exact one method declaration.

> Question: So, Now code is working when \*\*Object clone();\*\* method is not commented. So means this method is declared in interface Demo. But when we click on its implementation we move on Object class’s clone() method.  
When we comment clone() method and un-comment equals() method, then we get compile time error, interface is not FunctionalInterface. Why ?????? and why its functional interface with clone() method.

Please don’t say clone() is protected, what’s the matter if clone is protected in Object class. Please explain for me.

Thanks, sawai

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36888)

1. **nimisha says**

[SEPTEMBER 12, 2016 AT 4:23 AM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36299)

I didnt get how you are marking interfaces as funcyional or non functional on basis of signature. Please explain.

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36299)

1. **Vipul says**

[SEPTEMBER 2, 2016 AT 7:50 AM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36213)

What is numberTest here ??

return numbers.stream()  
.filter(NumberTest::isOdd) //Predicate is functional interface and  
.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it  
.filter(NumberTest::isLessThan11) // rather than anonymous inner classes  
.max(Comparator.naturalOrder())  
.map(i -> i \* i)  
.get();

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-36213)

* + **Ravi says**

[DECEMBER 24, 2016 AT 8:34 PM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-37106)

NumberTest is the name of the class containing the static methods which are referenced in the lambda expression.

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-37106)

1. **Hitendra Kumar says**

[AUGUST 6, 2015 AT 4:54 AM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-32917)

Good to see you here ! You remember me?

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-32917)

1. **Sam says**

[SEPTEMBER 12, 2014 AT 5:47 AM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-30111)

Nice tutorial. Just a little bug in your code for prime numbers:

//Declarative approach  
private static boolean isPrime(int number) {  
return number > 1  
&& IntStream.range(2, **number - 1**).noneMatch(  
index -> number % index == 0);  
}

I think you mean “range(2, number)” since the end is exklusive

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-30111)

* + [**Pankaj**](http://www.journaldev.com/)**says**

[SEPTEMBER 12, 2014 AT 6:50 AM](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-30114)

Hi Sam,

Thanks for pointing it out, i have corrected it.

[Reply](http://www.journaldev.com/2763/java-8-functional-interfaces#comment-30114)