# 03: ♦ Functional interfaces and Lambda expressions Q&A

Posted on November 8, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

The **functional interfaces** and **Lambda expressions** are going to make your code concise when coding with Java 8. These are the most awaited features among Java developers. If you understand the following examples, you will know what a lambda expression is & why functional interfaces were introduced in Java 8 to enable functional programming (**FP**).

Q1. What is a Lambda expression?
A1. Lambda expressions are **anonymous methods** which are intended to replace the bulkiness of anonymous inner classes with a much more compact mechanism.

Q2. What is a functional interface?
A2. The annotation **@FunctionalInterface** is introduced in Java 8. This annotation acts similarly to @Override by signaling to the compiler that the interface is intended to be a functional interface. A functional interface can only have **one method** declaration. The compiler will throw an error if the interface has multiple abstract method declarations.

Q2. What does a lambda expression entail?
A2. 2 parts.

## Part 1: The body of lambda expression.

```
1
2 (int a, int b) -> a + b;
3
```

where “a” and “b” are input arguments of type int, and the body of the lambda expression evaluates “a+b” and returns the result of type int. The input argument type declaration is optional. So, you can write the above example as shown below without the “int”.

```
1
2 (a, b) -> a + b;
3
```

## Part 2: The signature of lambda expression

Declared via a functional interface. A functional interface is a single method interface. The functional interface demonstrated here takes two input arguments of types integer and returns a result of type integer.

```
1
2 @FunctionalInterface
3 public interface Summable {
4     abstract int sum(int input1, int input2);
5 }
6
```



## As a Java Architect

Java architecture & design concepts interview Q&As with diagrams | What should

The “Summable” is the type of lambda expression “(a, b) -> a + b;”.

```
1
2 Summable sumType = (a, b) -> a + b;
3
```

Can be used as shown below:

```
1
2 public class LambdaAssignedToFunctionalInterface
3     public static void main(String[] args) {
4         Summable sumType = (a, b) -> a + b;
5         int result = sumType.sum(5, 6);
6         System.out.println("result=" + result);
7     }
8 }
9
```

**Outputs:**

```
1
2 result=11
3
```

Now, the same example with Generics included.

# Example 1:

**Signature:**

```
1
2 @FunctionalInterface
3 public interface Summable<T, U, R> {
4     public R sum(T input1, U input2); // abstract
5                                       // is optio
6 }
7
```

Where, **T**, and **U** are input arguments type and **R** is a result type.

The **lambda expression body** is assigned to the **lambda type signature** with Generics.

be a typical Java EE architecture?

## Senior Java developers must have a good handle on

open all | close all
- Best Practice (6)
- Coding (26)
- Concurrency (6)
- Design Concepts (7)
- Design Patterns (11)
- Exception Handling (3
- Java Debugging (21)
- Judging Experience In
- Low Latency (7)
- Memory Managemen
- Performance (13)
- QoS (8)
- Scalability (4)
- SDLC (6)
- Security (13)
- Transaction Managen

## 80+ step by step Java Tutorials

open all | close all
- Setting up Tutorial (6)
- Tutorial - Diagnosis (2
- Akka Tutorial (9)
- Core Java Tutorials (2
- Hadoop & Spark Tuto
- JEE Tutorials (19)
- Scala Tutorials (1)

```
1
2 public class LambdaAssignedToFunctionalInterface
3     public static void main(String[] args) {
4         Summable<Integer, Integer, Integer> sumTy
5         int result = sumType.sum(5, 6);
6         System.out.println("result=" + result);
7     }
8 }
9
```

A new package "**java.util.function**" is added with a number of functional interfaces to provide target types for lambda expressions and method references. E.g. Function, Consumer, IntConsumer, Predicate, Supplier, ToIntFunction, LongFunction, etc to name a few.

This means, in most cases you don't have to define your own Functional Interface like "Summabale". You can reuse the "java.util.function.**BiFunction**" functional interface that takes **any 2 input arguments** and **returns a result** as shown below:

```
1
2  import java.util.function.BiFunction;
3
4  public class LambdaAssignedToFunctionalInterface
5      public static void main(String[] args) {
6          BiFunction<Integer, Integer, Integer> su
7          int result = sumType.apply(5, 6);
8          System.out.println("result=" + result);
9      }
10 }
11
12
```

**Outputs:**

```
1
2 result=11
3
```

Google for the keywords "Grep code BiFunction" to see how "BiFunction" is implemented in JDK.

# Example 2:

Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

## Preparing for Java written & coding tests

open all | close all
♦ Complete the given
Can you write code? (
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

## How good are your...to go places?

open all | close all
Career Making Know-
Job Hunting & Resum

Using the existing Java **Comparator** interface and implementing your own implementation to compare two numbers.

**Step 1**: The functional interface from JDK package “java.util”. Note that as we mentioned earlier, a functional interface can only have one method declaration, and in the example below it is “int compare(T o1, T o2)”. The “boolean equals(Object obj);” is allowed because it is from the **java.lang.Object** class, which is an exception and is allowed. You can also have any number of **default** and **static** method implementations.

```
1
2  package java.util;
3
4  import java.io.Serializable;
5  import java.util.function.Function;
6  import java.util.function.ToIntFunction;
7  import java.util.function.ToLongFunction;
8  import java.util.function.ToDoubleFunction;
9  import java.util.Comparators;
10
11 @FunctionalInterface
12 public interface  Comparator<T> {
13
14    int compare(T o1, T o2);
15
16    boolean  equals(Object obj); //Object class r
17
18     default Comparator<T> reversed() {
19       return Collections.reverseOrder(this);
20     }
21
22     //many other default method implementations
23
24     public static <T extends Comparable<? super
25         return Collections.reverseOrder();
26     }
27
28     //many other static method implementations
29
30 }
31
```

**Step 2**: The lambda expression bod.

```
1  package com.java8.examples;
2
3  import java.util.Comparator;
4
5  public class CompareTest {
6
```

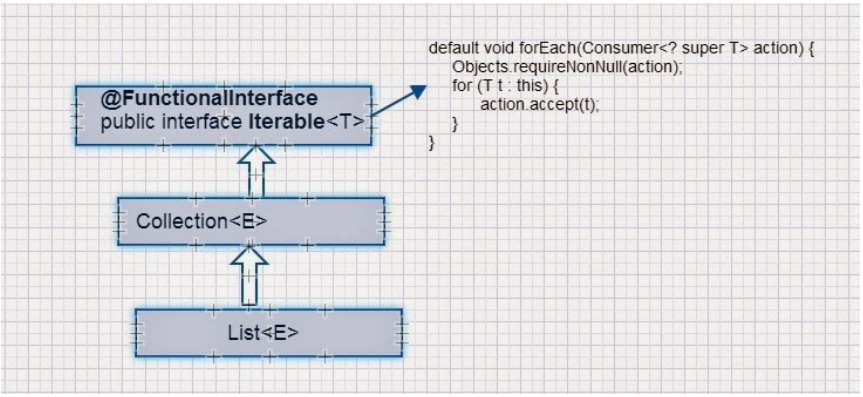
```
7   public static void main(String[] args) {
8      System.out.println(new CompareTest().compare
9   }
10
11  public int compare( int input1,  int input2) {
12     //(argument) -> (body)
13     //(input) -> (result)
14
15     Comparator cmp = (x, y) -> {
16        return (x < y) ? -1 : ((x > y) ? 1 : 0);
17     };
18
19     //invoke anonymous method
20     return cmp.compare(input1, input2);
21  }
22 }
23
```

The **output**:

```
1
2 -1
3
```

# Example 3:

It is very common to loop through a collection of objects and perform certain tasks like setting a variable, etc. The *Iterable* from which the other interfaces like *Collection*, *List*, etc has been annotated with ***@FunctionalInterface***.




As shown in the diagram, the *Iterable* also has a default method forEach(Consumer action) implemented to loop through the collection.

**Step 1**: Create a Person POJO object with fields, getters, and setters.

```
1   package com.java8.examples;
2
3  public class Person {
4
5     private String firstName;
6     private String surname;
7
8     public Person(String firstName) {
9         this.firstName = firstName;
10    }
11
12    public String getFirstName() {
13        return firstName;
14    }
15
16    public void setFirstName(String firstName) {
17        this.firstName = firstName;
18    }
19
20    public String getSurname() {
21        return surname;
22    }
23
24   public void setSurname(String surname) {
25       this.surname = surname;
26   }
27
28   @Override
29   public String toString() {
30       return "Person [firstName=" + firstName +
31   }
32 }
33
```

**Step 2**: Create a collection of *Person* objects, and ***forEach*** person, set the surname using the Lambda expression and anonymous method call. This is functional programming.

```
1   package com.java8.examples;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class PersonCollectionTest {
7
8   public static void main(String[] args) {
9      List<Person> persons = new ArrayList<>();
10     persons.add(new Person("John"));
11     persons.add(new Person("Peter"));
12
13     // Java 8: Lambda expression that sets surna
14     persons.forEach((p) -> {p.setSurname("Smith"
15
16     System.out.println(persons);
17  }
18
19 }
20
```

The output will be

```
1 [Person [firstName=John, surname=Smith], Person |
```

Now, what type is the lambda expression “(p) -> {p.setSurname(“Smith”);}”? in other words what is it its signature? Its signature is takes an input of type “Person” and return type is void. So this is of type

java.util.function.**Consumer<T>**;

This has an abstract method of:

```
1
2 void accept(T t);
3
```

So, it can be re-written as:

```
1   package com.java8.examples;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.function.Consumer;
6
7  public class PersonCollectionTest {
8
9   public static void main(String[] args) {
10     List<Person> persons = new ArrayList<>();
11     persons.add(new Person("John"));
12     persons.add(new Person("Peter"));
13
14     // Java 8: Lambda expression that sets surna
15     Consumer<Person> consumer = (p) -> {p.setSur
16
17     persons.forEach(consumer);
18
19     System.out.println(persons);
20  }
21
22 }
23
```

It can also be used as “consumer.**accept**(p);”

```
1
2  import java.util.function.Consumer;
```

```
3
4  public class PersonCollectionTest {
5
6   public static void main(String[] args) {
7      // Java 8: Lambda expression that sets surna
8      Consumer<Person> consumer = (p) -> {p.setSur
9
10     Person p = new Person("John");
11     consumer.accept(p);
12
13     System.out.println(p);
14  }
15
16 }
17
```

# Example 4:

The *Runnable* interface and *Thread* class to create new threads via Lambda expressions.

The Java API *Runnable* interface in Java 8 has been annotated with ***@FunctionalInterface***, which means you can only have 1 abstract method.

```
1 @FunctionalInterface
2 public interface Runnable {
3     public abstract void run();
4 }
5
```

```
1 public class Thread implements Runnable {
2    //.........skipped
3 }
4
```

Here is Lambda expression in action to create new threads. The Runnable functional interface takes no arguments as input, and returns nothing.

```
1
2
3  package com.java8.examples;
4
5  import java.util.concurrent.atomic.LongAdder;
6
7  public class ThreadTest {
8   private static LongAdder counter = new LongAdde
9
10  public static void main(String[] args) {
11
```

```
12    new Thread(() -> { counter.increment();
13                       System.out.println(Thread.c
14                     }).start();
15
16    Thread t2 = new Thread(() -> { counter.increme
17                       System.out.println(Thread.c
18                     });
19    t2.start();
20
21
22   }
23 }
24
25
```

The **output**:

```
1
2 Thread-0 count -- 1
3 Thread-1 count -- 2
4
```

The above code can be rewritten as shown below:

```
1
2  import java.util.concurrent.atomic.LongAdder;
3
4  public class ThreadTest {
5      private static LongAdder counter = new LongA
6
7      public static void main(String[] args) {
8
9          Runnable runnable1 = () -> {
10             counter.increment();
11             System.out.println(Thread.currentThr
12         };
13
14
15         Runnable runnable2 = () -> {
16             counter.increment();
17             System.out.println(Thread.currentThr
18         };
19
20         Thread t1 = new Thread(runnable1);
21         Thread t2 = new Thread(runnable2);
22
23         t1.start();
24         t2.start();
25     }
26 }
27
```

# Example 5:

The free variables in lambda expressions are not thread-safe.
A lambda expression has 3 aspects:

1. Parameters
2. A block of code or body
3. Values for the **free variables**. These are the variables that are not parameters, and not defined inside a closure.

The anonymous inner classes can only access variables defined outside if they are marked final. Otherwise you will get a compile error. The lambda expression has relaxed the requirement of variables being final, but the “free variables” that you use need to be either **final** or **effectively final**.

```
1  package com.java8.examples;
2
3  import java.util.concurrent.atomic.LongAdder;
4
5  public class ThreadTest {
6    private static LongAdder counter = new LongAdd
7
8    public static void main(String[] args) {
9
10    int counterLocal = 0;  //local free variable
11
12
13    new Thread(() -> {
14       counter.increment();
15
16       //local variables referenced from a lambd
17       counterLocal++; //Line A: illegal. compile
18
19       System.out.println(counterLocal); // Line
20       int a = counterLocal; //Line C: Ok, as cou
21
22          System.out.println(Thread.currentThread
23    }).start();
24
25
26    }
27 }
28
```

The above code gives compile error due to Line A. If you comment out Line A, the above code will compile, and Line B and Line C are ok as they use the variable *counterLocal* as read only and it is **effectively final**.

So, mutating free variables in a lambda expression is not thread-safe. The prohibition against mutation only holds true

for local variables as explained above. You can still use an instance or static variable, but the compiler won't warn you, but you can have thread-safety issues.

# Example 6:

Understanding scope of "**this**" keyword.

Will the *toString( )* method invocation shown below invokes *ThreadTest's* toString( ) method or *Thread* class's *toString( )* method?

```
1  package com.java8.examples;
2
3  import java.util.concurrent.atomic.LongAdder;
4
5  public class ThreadTest {
6
7   public static void main(String[] args) {
8         new  ThreadTest().execute();
9   }
10
11  public void execute() {
12   new Thread(() -> {
13           System.out.println(this.toString());
14    }).start();
15  }
16
17
18  @Override
19  public String toString() {
20   return "ThreadTest class toString()";
21  }
22 }
23
```

**Output** is:

```
1
2 ThreadTest class toString()
3
```

So, there is nothing special about using the "This" keyword in lambda expressions. The scope of lambda expression is nested inside the *execute( )* method, and this has the same meaning anywhere inside the execute method.

# Popular Posts

♦ 11 Spring boot interview questions & answers

**861 views**

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

**829 views**

18 Java scenarios based interview Questions and Answers

**448 views**

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

**407 views**

♦ 7 Java debugging interview questions & answers

**311 views**

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

**303 views**

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

**294 views**

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

**288 views**

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

**263 views**

8 Git Source control system interview questions & answers

**215 views**

**Bio** **Latest Posts**

**About** Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold

35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ Java 8: Does “Optional” class alleviate the pain of ubiquitous NullpointerException?

04: ♥♦ Top 6 tips to transforming your thinking from OOP to FP with examples ›

**Posted in** FP, Java 8, member-paid

# Leave a Reply

Logged in as geethika. Log out?

**Comment**

Post Comment

# Empowers you to open more doors, and fast-track

### Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

### Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

## Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category