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

Posted on November 8, 2014 by Arulkumaran Kumaraswamipillai — 5 Comments ↓

One needs to get used to the transformation from **imperative programming** to **functional programming**. You like it or not, you will be using functional programming in Java, and interviewers are going to quiz you on functional programming. Fortunately, Java is not a fully functional programming language, and hence one does not require the full leap to functional programming. Java 8 supports both imperative and functional programming approaches.

Q1. What is the difference between imperative and declarative programming paradigms?

A1. **Imperative** (or procedural) programming: is about

defining the computation how to do something in terms of statements and state changes, and as a result what you want to happen will happen.

**Declarative programming**: is about declaratively telling what you would like to happen, and let the library or functions figure out how to do it. SQL, XSLT and regular expressions are declarative languages.

Q2. Does functional programming use imperative or declarative approach?
A2. Functional programming is a form of declarative programming, where **functions are composed of other functions** — g(f(x)) where g and f are functions. An output of one function becomes the input for the composing function. A typical example of functional programming, which you may have used is transforming an XML document using XSLT to other forms. The composable and isolated XSL style sheets are used for transformation.

Q3. What are the differences between OOP and FP?
A3.

| | OOP | FOP |
|---|---|---|
| **focus** | To solve problems, OOP developers design class hierarchies, focus on proper encapsulation, and think in terms of class contracts (i.e. design by contract). The behavior and state of object types are of paramount importance, and language features, such as classes, interfaces, inheritance, and polymorphism, are provided to address these concerns. | To solve computational problems by evaluating pure functions to transform a collection of data. Functional programming avoids state and mutable data, and instead emphasizes on the application and composition of functions. |
| **flow control** | loops, conditions, and method calls | function calls and recursion |
| **state changes** | Necessary | No state changes. **Effectively final variables** are used in Java |
| **order of execution** | High importance | low importance |

In OOP, x = x+ 5 makes sense, but in mathematics or functional programming, you can't say x = x + 5 because if x were to be 2, you can't say that 2 = 2 + 5. In functional programming you need to say f(x) -> x + 5.

## #1. Focus:

**OOP** focuses on solving business problems by designing classes, interfaces, and contracts. The behavior and state are

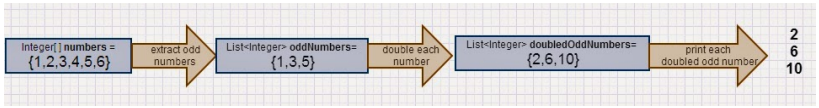


## As a Java Architect

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

very important to OOP with OO concepts scubas polymorphism, inheritance, and encapsulation.

**FP** focuses on computational problems by evaluating functions to transform a collection of data. FP avoids state and mutable data, and instead focuses on the composition and application of functions.




Java functional programming focusing on transforming data

## #2. Flow Control:

**OOP** uses loops, conditions, and method calls. Order of execution is very important.

**FP** uses function calls and recursion. Order of execution is less important.

Q4. Where does functional programming shine?
A4. **Example:** Here is an example written functional programming to extract odd numbers from a given list of numbers and then double each odd number and print each of them.

**Functional programming** using the Java 8 lambda expressions.

Functional programming with good set of libraries can cut down lot of fluff and focus on just transformations. In other words, just tell what you would like to do.

```
1
2  package com.java8.examples;
3
4  import java.util.Arrays;
5
6  public class NumberTest {
7
8   public static void main(String[] args) {
```

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)

```
 9    Integer[] numbers = {1,2,3,4,5,6};
10
11    Arrays.asList(numbers)   //convert to least
12          .stream()          //stream
13          .filter((e) -> (e % 2 != 0))   // extract
14          .map((e) -> (e * 2))           // double
15          .forEach(System.out::println); // print e
16     }
17  }
18
19
```

**Output:**

```
1
2 2
3 6
4 10
5
```

**Shining moment 1:** The FP has much improved readability and maintainability because each function is designed to accomplish a specific task for given arguments. OOP or procedural programming to accomplish the same will require for loops and will be more verbose.

**Shining moment 2:** A functional program can be easily made to run in parallel using the “fork and join” feature added in Java 7. To improve performance of the above code. All you have to do is use .parallelStream( ) instead of .stream( ).

```
1
2  import java.util.Arrays;
3
4  public class NumberTest {
5
6     public static void main(String[] args) {
7       Integer[] numbers = {1,2,3,4,5,6};
8
9       Arrays.asList(numbers)
10         .parallelStream( )      //use fork and j
11         .filter(NumberTest::isOddNumber)
12         .map(NumberTest::doubleIt)
13         .peek(x -> System.out.println(Thread.cur
14         .count(); // a terminal operation is req
15                   // if count() is omitted, noth
16    }
17
18    private static boolean isOddNumber(int input)
19        return input % 2 != 0;
20    }
21
22    private static int doubleIt(int input) {
23        return input * 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

```
24     }
25 }
26
```

```
1
2 ForkJoinPool.commonPool-worker-2 processed 10
3 ForkJoinPool.commonPool-worker-1 processed 6
4 ForkJoinPool.commonPool-worker-3 processed 2
5
```

**Note**: Also, the Java 8 **CompletableFuture** is enabled for functional programming to write more elegant asynchronous code in Java. Look at ForkJoinPool, ExecutorService & CompletableFuture Q&A.

Learn more about terminal vs intermediate operations with diagrams.

**Shining moment 3:** The code is also easier to refactor as shown below. If you want to switch the logic to double it first and then extract the odd numbers, all you have to do is swap the **.filter** call with **.map** call.

```
1
2 Arrays.asList(numbers)
3         .parallelStream()
4         .map(NumberTest::doubleIt)
5         .filter(NumberTest::isOddNumber)
6         .forEach(System.out::println);
7
8
```

**Shining moment 4:** Easier testing and debugging. Because pure functions can more easily be tested in isolation, you can write test code that calls the pure function with typical values, valid edge cases, and invalid edge cases. In the above example, I was using the Java 8 library that was well tested with confidence. I will only have to provide unit tests for the static functions “boolean isOddNumber(int number)” and “int doubleIt(int number)” that provide the logic.

Having said this, OO programming and functional programming can co-exist. Both have their strengths and weaknesses. So, both compliment each other.

Q5. What are the characteristics of functional programming you need to be familiar with?

A5.

1. A focus on what is to be computed rather than how to compute it.
2. Function Closure Support
3. Higher-order functions
4. Use of recursion as a mechanism for flow control
5. Referential transparency
6. No side-effects

Let's look at these one by one:

# 1. A focus on what is to be computed rather then how to compute it

```
1  Integer[] numbers = {1,2,3,4,5,6};
2
3  //focusses on what to do as opposed to how to do
4  //no fluff like for loops, mutations, etc
5  Arrays.asList(numbers)
6          .stream()
7          .filter((e) -> (e % 2 != 0))
8          .map((e) -> (e * 2))
9          .forEach(System.out::println);
10
11
```

Extract odd numbers, and multiply each by 2, and the print the result.

# 2. Function closure support

In order to create closures, you need a language where the function type is a 1st class citizen, where a function can be assigned to a variable, and then passed around like any other variables like a string, int or boolean. closure is basically a snapshot of the stack at the point that the lambda function is

created. Then when the function is re-executed or called back the stack is restored to that state before executing the function.

The **java.util.function** package provides a number of functional interfaces like Consumer, Function, etc to define closures or you can define your own functional interfaces.

```
1  package com.java8.examples;
2
3  import java.util.function.Function;
4
5  public class ClosureTest {
6
7   public static void main(String[] args) {
8
9    //closure 1 that adds 5 to a given number
10   Function<Integer, Integer> plus5 = (i) -> (i+5
11   //closure 2 that times by 2 a given number
12   Function<Integer, Integer> times2 = (i) -> (i*
13   //closure 3 that adds 5 and then multiply the
14   Function<Integer, Integer> plus5AndThenTimes2
15   //closure 4 that times by 2 and then adds 5
16   Function<Integer, Integer> times2AndThenplus5
17
18         //callback or execute closure
19   //functions plus5, times2, etc can be passed a
20   System.out.println("9+5=" + execute(plus5, 9))
21   System.out.println("9*2=" + execute(times2, 9)
22   System.out.println("(9+5)*2=" + execute(plus5A
23   System.out.println("9*2+5=" + execute(times2An
24
25  }
26
27  //functions can be used as method parameters
28  private static Integer execute(Function<Integer
29   return function.apply(number); //execute the f
30  }
31 }
32
```

**Output**:

9+5=14
9*2=18
(9+5)*2=28
9*2+5=23

In pre Java 8, you can use a**nonymous inner classes** to define closures. In Java 8, lambda operators like (i) -> (i+5) are used to denote anonymous functions.

### Is currying possible in Java 8?

Currying (named after Haskell Curry) is the fact of evaluating function arguments one by one, producing a new function with one argument less on each step.

Java 8 still does not have first class functions, but currying is “practically” possible with verbose type signatures, which is less than ideal. Here is a very simple example:

```
1  package com.java8.examples;
2
3  import java.util.function.Function;
4
5  public class CurryTest {
6
7   public static void main(String[] args) {
8    Function<Integer,Function<Integer,Integer>> ad
9
10   Function<Integer,Integer> addOne = add.apply(1
11   Function<Integer,Integer> addFive = add.apply(
12   Function<Integer,Integer> addTen = add.apply(1
13
14   Integer result1 = addOne.apply(2); // returns
15   Integer result2 = addFive.apply(2); // returns
16   Integer result3 = addTen.apply(2); // returns
17
18   System.out.println("result1 = "  + result1);
19   System.out.println("result2 = "  + result2);
20   System.out.println("result3 = "  + result3);
21
22  }
23 }
24
```

The **output** is

result1 = 3
result2 = 7
result3 = 12

# 3. Higher order functions

In mathematics and computer science, a **higher-order function** (aka functional form) is a function that does at least one of the following:

1. takes one or more functions as an input — for example g(f(x)), where f and g are functions, and function g composes the function f.
2. **outputs a function** — for example, in the code above ***plus5*** and *plus2* outputs a function

```
1 Function<Integer, Integer> plus5 = (i) -> (i+5);
2  //closure 2 that times by 2 a given number
3  Function<Integer, Integer> times2 = (i) -> (i*2)
4
```

also

```
1 Function<integer integer> plus5AndThenTimes2 = pl
2
```

outputs another function, where the **plus5** function takes **plus2** function as an input.

# 4. Use of recursion as a mechanism for flow control

Java is a stack based language that supports **reentrant (**a method can call itself) methods. This means recursion is possible in Java. Using recursion you don't need a mutable state, hence the problems can be solved in a much simpler fashion compared to using an iterative approach with a loop.

```
1  package com.java8.examples;
2
3  import java.util.function.Function;
4  import java.util.function.IntToDoubleFunction;
5
6  public class RecursionTest {
7
8      static class Recursive<I> {
9          public I func;
10     }
11
12     static Function<Integer, Double> factorial =
13         Recursive<IntToDoubleFunction> recursive
14         recursive.func = n -> (n == 0) ? 1 : n
15                 * recursive.func.applyAsDouble(n
16
17         return recursive.func.applyAsDouble(x);
18     };
19
```

```
20      public static void main(String[] args) {
21
22          Double result = factorial.apply(3);
23          System.out.println("factorial of 3 = " +
24      }
25  }
26
```

Recursion is used in the Lambda expression to calculate the factorial. This is not very straight-forward, and here are the key points to understand the above code.

**1)** In java 8, you have Consumer<T> and Function<T, R> interfaces where a **Consumer** takes an object input and returns nothing and a **Function** takes an Object input and returns a result of type object.

"factorial" is a "**Function**" that takes an **input** of type "Integer" and result of type "Double". The output needs to be a double because the factorials can get to very large numbers and int and long are not appropriate as this will lead to data overflow.

**2)** In lambda, you can't specify a recursion as shown below as you will get a compile error of "**The method factorial(int) is undefined**"

```
1
2  static Function<Integer, Double> factorial = x ->
3      return (x == 0) ? 1.0 : x * factorial(x-1);
4  };
5
```

The lambda expression and anonymous classes capture local variables by value when they are created. Therefore, it is impossible for them to refer to themselves by capturing a local variable as the value for pointing to itself does not exist yet at the time it is being created.

**3)** So, to overcome the above problem, let's create a generic helper class that wraps the variable of the functional interface type. The functional interface type is "**IntToDoubleFunction**", which converts an int to double.

**4)** The “applyAsDouble” method applies **this** function to the given argument.

# 5. Referential Transparency

Referential transparency is a term commonly used in **functional programming**, which means that given a function and an input value, you will always receive the same output. There is no external state used in the function. In other words, a referentially transparent function is one which only depends on its input. The ***plus5*** and ***times2*** functions shown above are referentially transparent.

A function that reads from a text file and prints the output is not referentially transparent. The external text file could change at any time so the function would be referentially **opaque**

# 6. No side-effects

One way to do programming without side effects is to use only immutable classes. In real world, you try to minimize the mutations, but can’t eliminate them. Lambdas can refer to local variables that are not declared final but are never modified. This is known as “**effectively final**“.
When you are using methods like **reduce** on a stream, you need to be aware of the side effects due to parallelism. For example, the following code will have no side effects when run in serial mode.

```
1 public static void main(String[] args) {
2  Integer[] numbers = {10, 6};
3  Integer result = Arrays.asList(numbers).stream()
4                         .reduce(20, (a,b) -> (a - b
5
6  System.out.println(result);
7 }
8
```

**Outputs**:

4

It starts with 20, and then subtracts 10 and 6. **20 – 10 – 6 = 4**;
But if you run it again in parallel mode as shown below

```
1 public static void main(String[] args) {
2  Integer[] numbers = {10, 6};
3  Integer result = Arrays.asList(numbers).parallel
4                         .reduce(20, (a,b) -> (a
5
6  System.out.println(result);
7 }
8
```

The **output** will be:

-4

**What happened here?** 10 – (20 – 6) = -4;

This means the **reduce** method on a stream produce side effects when run in parallel for **non-associative** operations.

**Q**. What is an associative property?
**A**. Associative operations are operations where the order in which the operations were performed does not matter. For example.

**Associative**:

3 + (2 + 1) = (3+ 2) + 1
3 * (2 * 1) = (3 * 2) * 1

**Non-associative**:

3 – (2 – 1) != (3 – 2) – 1
(4/2)/2 != 4/(2/2)

Pure functions are functions with no side effects. In computers, **idempotence** means that applying an operation once or applying it multiple times has the same effect. For example

**Idempotent operations**

- Multiplying a number by zero. No matter how many times you do it, the result is still zero.
- Setting a boolean flag. No matter how many times you do it, the flag stays set.
- Deleting a row from a database with a given ID. If you try it again, the row is still gone.
- Removing an element from a collection.

In real life where you have concurrent operations going on, you may find that operations you thought were idempotent cease to be so. For example, another thread could unset the value of the boolean flag.

## Java 8 functional programming examples

**1.** 7 Java FP (lambda expressions) real life examples in wrangling normal & big data

**2.** 19 Java 8 Functional Programming (i.e. FP) interview Q&As with examples

## 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

## 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.

**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.

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

Top 6 Java 8 features you can start using now ›

**Posted in** FP, Java 8, OOP & FP Essentials

**Tags:** TopX

## 5 comments on "04: ♥♦ Top 6 tips to transforming your thinking from OOP to FP with examples"

sandymehta says:

October 14, 2015 at 8:18 am

Hi Arun

The following program does not compile and the error message for factorial variable declaration is "cannot reference a field before it is defined"

```
package com.java8.examples;

import java.util.function.Function;

public class RecursionTest {

static Function factorial =
x -> {
return ( x == 0) ? 1 : x * factorial.apply(x-1); // compiler
error
};

public static void main(String[] args) {

int result = factorial.apply(3);
System.out.println("factorial of 3 = " + result);
}
}
```

Reply

Arulkumaran Kumaraswamipillai says:

October 19, 2015 at 11:02 pm

FIXED. Thanks for pointing that out.

Reply

Kiran says:

February 19, 2015 at 12:01 pm

Very nice explaination

Reply

Arulkumaran says:

February 19, 2015 at 10:28 pm

Thanks Kiran.

Reply

Arulkumaran says:

February 25, 2015 at 3:33 pm

Thanks Kiran. You will also like my latest post on Streams.

Reply

# 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

# © Disclaimer