**OOP to FP**

<https://www.java-success.com/java-8-transforming-thinking-oop-fop-java-8-examples/>

Functional programming is prevalent in data engineering (i.e. Spark programming, etc) roles. A must know for Big Data Engineers and Analysts.

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 is Unix pipe lines where output of one function becomes input of next.



|  |  |
| --- | --- |
| 1 | ls -l | grep “Dec” | Sort +4n | more |

In FP you do:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Integer[] numbers = {1,2,3,4,5,6};      Arrays.asList(numbers)              //convert to least         .stream()                      //stream         .filter((e) -> (e % 2 != 0))   // extract only odd numbers 1, 3, 5         .map((e) -> (e \* 2))           // double the odd numbers 2, 6, 10         .forEach(System.out::println); // print each doubled number.  } |

**No more** “for” loops and mutating (i.e. changing states of) variables. Each functions like filter, map, etc return a **new copy** of the data without mutating the input data.

Functions can be assigned to immutable variables and passed around as args:



|  |  |
| --- | --- |
| 1  2  3 | val a = f(x)  val b = g(a) |

Which means we can substitute f(x) for a:



|  |  |
| --- | --- |
| 1  2 | val b = g(f(x)) |

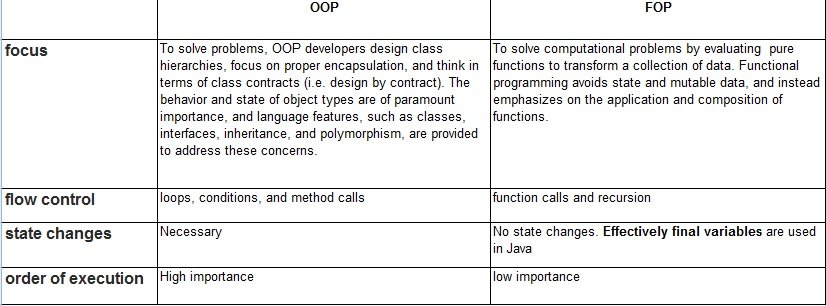
Q3. Where would you use FP?  
A3. As IT professionals, we need to use the right paradigm or tool for the right job. In Java, OOP, FP, and AOP can co-exist. These paradigms compliment each other. If used properly, FP is very powerful at

**1)** Solving problems that require lots of concurrency & parallelism. FP favors immutability, hence easier to parallelize. FP shines in efficient processing of BigData.

**2)** Applying math heavy algorithms or a series of transformations. For example, solving a problem that can be decomposed into a series of mathematical operations. A very simple example would be to take a list of numbers, filter out only the odd numbers, double the resulting odd numbers, compute the sum, and so on.

**3)** Creating Domain Specific Languages (i.e. DSLs). A DSL is a computer language targeted at solving a particular kind of problem and it is not planned to solve problems outside of its domain. DSLs have become a valuable part of the Groovy language.

Q4. What are the differences between OOP and FP?  
A4.

[](https://www.java-success.com/wp-content/uploads/2014/11/OOP_vs_FOP.jpg)

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. All about

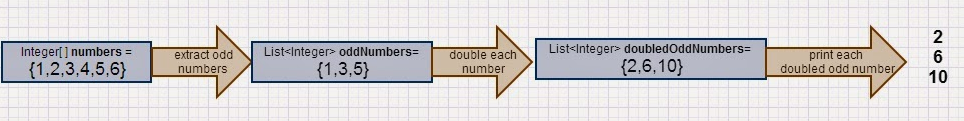
**OOP** is all about objects. Objects can be stored in variables & collections, objects can be passed around in method arguments, objects can be composed, etc. The objects are the first class citizens.

**FP** is all about functions. Functions can be stored in a variables & collections, functions can be passed as arguments to functions which can return a function, etc. Functions manipulating other functions are called **higher-order** functions.

#### #2. Focus:

**OOP** focuses on solving business problems by designing classes, interfaces, and contracts. The behavior and state are very important to OOP, and applies OO concepts such as polymorphism, inheritance, and encapsulation.

**FP** focuses on computational problems by evaluating functions to transform a collection of data by focusing on the composition and application of functions.

[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-13-at-12.08.38-PM.png)

Java functional programming focusing on transforming data

#### #3. Flow Control:

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

**FP** does not like loops as the loops are for state change constructs. FP uses function calls and recursion. Order of execution is less important.

#### #4. Mutability:

**OOP** uses loops to mutate state. For example, increases the employee salary by 5% as shown below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};    List<employee> inputList = Arrays.asList(employees);    //state change  for (Employee employee : inputList) {      employee.setSalary(employee.getSalary() \* 1.05);  }    //original list is mutated  System.out.println(inputList); //has side-effects  </employee> |

**FP** favours immutability. Pure functions must always return the same output for a given input. This is called **idempotency** and **referential transparency**. In a real world, programs need to alter values by taking some data as input, and output some results. FP avoids mutation by returning **new data** instead.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};    List<employee> inputList = Arrays.asList(employees);  //apply 5% salary increase to the given input data  List<employee> newMutatedList = inputList.stream()                                    .map(x -> new Employee(x.getName(), x.getSalary() \* 1.05)) // returns a new list                                            .collect(Collectors.toList());    System.out.println(inputList); //[Employee [name=John, salary=45000.0], Employee [name=Peter, salary=65000.0], Employee [name=Sam, salary=85000.0]]  System.out.println(newMutatedList); //[Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [name=Sam, salary=89250.0]]  </employee></employee> |

The above code might seem inefficient, but there are techniques like lazy evaluation to optimize this practice of duplicating data. The real benefit of immutability is all about solving problems that require concurrency (aka parallelism) as shown below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};    List<employee> inputList = Arrays.asList(employees);    List<employee> newMutatedList = inputList.stream().parallel()                                   .map(x -> new Employee(x.getName(), x.getSalary() \* 1.05)) // returns a new list                                           .collect(Collectors.toList());    System.out.println(inputList); //[Employee [name=John, salary=45000.0], Employee [name=Peter, salary=65000.0], Employee [name=Sam, salary=85000.0]]  System.out.println(newMutatedList); //[Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [name=Sam, salary=89250.0]]  </employee></employee> |

A pragmatic FP as opposed to a purist FP can tolerate functions with side-effects, but avoid them as much as possible. For example, “forEach” function is a functional version of the old for loop. It should be used only to produce side-effects for each item. For example, issue a RESTful request, serialize an item, create new elements in a DOM, etc. The above salary increase code can be coded with side-effects as shown below:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};    List<employee> inputList = Arrays.asList(employees);    inputList.stream().parallel()            .forEach(x -> x.setSalary(x.getSalary() \* 1.05));    //original list is mutated  System.out.println(inputList); [Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [name=Sam, salary=89250.0]]  </employee> |

Document any side effects and isolate them to specific modules.

Q5. Where does functional programming shine?  
A5.**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  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.java8.examples;    import java.util.Arrays;    public class NumberTest {    public static void main(String[] args) {    Integer[] numbers = {1,2,3,4,5,6};      Arrays.asList(numbers)   //convert to least         .stream()          //stream         .filter((e) -> (e % 2 != 0))   // extract only odd numbers 1, 3, 5         .map((e) -> (e \* 2))           // double the odd numbers 2, 6, 10         .forEach(System.out::println); // print each doubled number.  }  } |

**Output:**



|  |  |
| --- | --- |
| 1  2  3  4  5 | 2  6  10 |

**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  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.util.Arrays;    public class NumberTest {       public static void main(String[] args) {       Integer[] numbers = {1,2,3,4,5,6};         Arrays.asList(numbers)          .parallelStream( )      //use fork and join thread pool introduced in Java 7          .filter(NumberTest::isOddNumber)          .map(NumberTest::doubleIt)          .peek(x -> System.out.println(Thread.currentThread().getName() + " processed " + x))          .count(); // a terminal operation is required as the intermediate operations are lazily evaluated.                    // if count() is omitted, nothing gets processed     }       private static boolean isOddNumber(int input) {         return input % 2 != 0;     }       private static int doubleIt(int input) {         return input \* 2;     }  } |



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

**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](https://www.java-success.com/10-%E2%99%A6-executorservice-vs-forkjoin-future-vs-completablefuture-interview-qa/).

Learn more about [terminal vs intermediate operations with diagrams](https://www.java-success.com/java-8-streams-lambdas-intermediate-vs-terminal-ops-lazy-loading-simple-examples/).

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

**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 than how to compute it



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

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  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  32 | package com.java8.examples;    import java.util.function.Function;    public class ClosureTest {    public static void main(String[] args) {      //closure 1 that adds 5 to a given number    Function<integer, integer=""> plus5 = (i) -> (i+5);    //closure 2 that times by 2 a given number    Function<integer, integer=""> times2 = (i) -> (i\*2);    //closure 3 that adds 5 and then multiply the result by 2    Function<integer, integer=""> plus5AndThenTimes2 = plus5.andThen(times2);    //closure 4 that times by 2 and then adds 5    Function<integer, integer=""> times2AndThenplus5 = times2.andThen(plus5);            //callback or execute closure    //functions plus5, times2, etc can be passed as arguments    System.out.println("9+5=" + execute(plus5, 9));    System.out.println("9\*2=" + execute(times2, 9));    System.out.println("(9+5)\*2=" + execute(plus5AndThenTimes2, 9));    System.out.println("9\*2+5=" + execute(times2AndThenplus5, 9));    }    //functions can be used as method parameters  private static Integer execute(Function<integer, integer=""> function, Integer number){    return function.apply(number); //execute the function  }  }  </integer,></integer,></integer,></integer,></integer,> |

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

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  2  3  4 | Function<integer, integer=""> plus5 = (i) -> (i+5);  //closure 2 that times by 2 a given number  Function<integer, integer=""> times2 = (i) -> (i\*2);  </integer,></integer,> |

also



|  |  |
| --- | --- |
| 1  2 | Function<integer integer=""> plus5AndThenTimes2 = plus5.andThen(times2);  </integer> |

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  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 | package com.java8.examples;    import java.util.function.Function;  import java.util.function.IntToDoubleFunction;    public class RecursionTest {        static class Recursive<i> {          public I func;      }        static Function<integer, double=""> factorial = x -> {          Recursive<inttodoublefunction> recursive = new Recursive<inttodoublefunction>();          recursive.func = n -> (n == 0) ? 1 : n                  \* recursive.func.applyAsDouble(n - 1);            return recursive.func.applyAsDouble(x);      };        public static void main(String[] args) {            Double result = factorial.apply(3);          System.out.println("factorial of 3 = " + result);      }  }  </inttodoublefunction></inttodoublefunction></integer,></i> |

*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  3  4  5 | static Function<integer, double=""> factorial = x -> {      return (x == 0) ? 1.0 : x \* factorial(x-1);  };  </integer,> |

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

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

*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.*