Register | Login | Logout | Contact Us

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Interview › Core Java Interview Q&A › FP › 01: ♦ 19 Java 8 Functional Programming (i.e. FP) interview questions and answers

01: ♦ 19 Java 8 Functional Programming (i.e. FP) interview questions and answers

Posted on December 1, 2015 by Arulkumaran Kumaraswamipillai

Good handle on Java OOP concepts and FP concepts are very important NOT only from writing quality code perspective, but also from job interviews perspective to be able to do well in written tests & technical interviews.

Q1. Can you explain your understanding of Functional Programming (FP)?
A1. In a very simplistic approach FP means:

#1 Programming without assignments: For example, in imperative style programming like OOP, you can say “x = x + 5”, which is an **assignment**, but in **mathematical** or **functional** programming you need to say f(x) -> x + 5. If, x were to be 2, it is mathematically incorrect to say “2 = 2 + 5”. In imperative style you are assigning a new value of 7 to x by saying “x = x + 5”. So, imperative programming has state and you can assign a new state. FP treats computation as the evaluation of mathematical functions and avoids storing state and mutating data.

#2 FP focuses on WHAT to be done, and NOT on HOW to be done: For example, in the ‘functional style loop’ you’ll focus only on the action of what to do on each element and you don’t have to concentrate on how to go through each element. The “forEach” function will take care of it.

Here is a simple example that shows imperative (e.g. OOP) and functional styles to loop through 1 to 10, square each number, and print squared numbers.

```


package com.mytutorial;

import java.util.stream.IntStream;

public class WorkWithNumbers {

    public static void main(String[] args) {
        WorkWithNumbers wwn = new WorkWithNumber
        wwn.printSquaresUptoTenImperativeStyle()
        System.out.println(); // newline
        wwn.printSquaresUptoTenFunctionalStyle()
    }

    //Imperative style
    public void printSquaresUptoTenImperativeSty
        for (int i = 1; i <= 10 ; i++) {
            System.out.print(i*i + " ");
        }
    }

    //FP style
    public void printSquaresUptoTenFunctionalSty
        IntStream.rangeClosed(1, 10)          //
                 .map(i -> i * i)             //
                 .forEach(i -> {              //
                      System.out.prin
                 });
    }

```

As a Java Architect

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

```
}
```

printSquaresUptoTenImperativeStyle

The variable “i” takes new value in each iteration. This is assignment as described in **#1**. The “for (int i = 1; i <= 10 ; i++)" defines "HOW to" go through each element as discussed in **#2**.

printSquaresUptoTenFunctionalStyle

1) Consists of 3 functions: “rangeClosed”, “map”, and “forEach”. The **mutable state** is avoided, and the output of a function is exclusively dependent on the values of its inputs. This means that if we call a function x amount of times with the same parameters we’ll get exactly the same result every time.

2) The “rangeClosed” function takes two arguments “startIclusive” and “endInclusive” to produce a range of integers (i.e 1 to 10).

3) The “map” function transforms a Collection, List, Set or Map. By using a “map” function you can apply a predefined function or a user defined function. You can not only use a lambda expression (e.g. i -> i * i), but also you can use a method reference (e.g System.out::println).

4) The “forEach” function loops through each element and performs the “action” of printing each squared integers. Focuses on what to do as per **#2**. Does not have to worry about “how to” iterate as “forEach” function will take care of the iteration logic.

Q2. Can you list the differences between OOP and FP?
A2. Top 6 tips to transforming your thinking from OOP to FP with examples

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 I
- ⊞ 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)

Q3. What are the 6 characteristics of FP you need to be familiar with?
A3.

1) A focus on what is to be computed rather then 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

Each of the above is explained in detail with examples at Top 6 tips to transforming your thinking from OOP to FP with examples

Q4. What do you understand by the terms streams, lambdas, intermediate vs terminal ops, and lazy loading with regards to FP?
A4. A **stream** is an infinite sequence of consumable elements (i.e a data structure) for the consumption of an operation or iteration.

An **Intermediate** operation like "map" produces another stream, whereas a "Terminal" operation like "forEach" produces an object or an output that is not a stream.

Intermediate operations are **lazy operations**, which will be executed only after a terminal operation is called. For example, when you call the intermediate operation ".map(i -> i * i)" the lambda body isn't executed until after the terminal operation ".forEach(i -> { System.out.print(i + " ");});" is called.

Learn more in detail with examples & diagrams: Java 8 Streams, lambdas, intermediate vs terminal ops, and lazy loading with simple examples

Q5. Why is it essential to understand lazy loading with regards to FP?

Preparing for Java written & coding tests

How good are your…to go places?

A5. This is essential to understand from the viewpoint of adding break points in your IDE for debugging purpose.

Q6. What is a functional interface?
A6. java.lang.Runnable, java.util.Comparator, and java.util.concurrent.Callable are Single Abstract Method interfaces (SAM Interfaces). For example, java.lang.Runnable has a single abstract method

```

package java.lang;

public interface Runnable {
    public abstract void run();
}

```

and these methods were called by anonymous classes like

```

    new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("Running in a new thre
      }
    }).start();

```

Functional interfaces introduced in Java 8 are recreation of SAM(Single Abstract Method) interface to allow functional programming with lambda expressions by adding an annotation **@FunctionalInterface** which can be used for compiler level errors when the interface you have annotated is not a valid Functional Interface.

```

  @FunctionalInterface
  public interface Summable {
      public int sum(int input1, int input2); //a
  }

```

Note: The method "sum" is an abstract method as the keyword "abstract" is optional in an interface.

Now, the following “**lambda expression**” is assignable to the functional interface type Summable.

```

Summable sumType = (a, b) -> a + b;

```

So, the **lambda expression** has two parts:

Part 1: The **body** of the expression denoted by “(a, b) -> a + b;”. This is equivalent to an anonymous method. aka no name method.

Part 2: The **signature** of the lambda expression via a functional interface. A functional interface is a single method interface. The functional interface takes two input arguments of types integer and returns a result of type integer.

```

public class LambdaAssignedToFunctionalInterfaceT
    public static void main(String[] args) {
        Summable sumType = (a, b) -> a + b;
        int result = sumType.sum(5, 6);
        System.out.println("result=" + result);
    }
}

```

Outputs:

```

result=11

```

Now, the same example with Generics included.

```

@FunctionalInterface
public interface Summable<T, U, R> {
    public R sum(T input1, U input2); // abstract
                                      // is optio
}

```

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

```

public class LambdaAssignedToFunctionalInterfaceT
    public static void main(String[] args) {
        Summable<Integer, Integer, Integer> sumTy
        int result = sumType.sum(5, 6);
        System.out.println("result=" + result);
    }
}

```

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 2 input arguments and returns a result as shown below:

```

import java.util.function.BiFunction;

public class LambdaAssignedToFunctionalInterface
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> su
        int result = sumType.apply(5, 6);
        System.out.println("result=" + result);
    }
}


```

Outputs:

```

result=11

```

Refer to:

1) Java 8 using the Predicate functional interface.

2) Java 8 API examples using lambda expressions and functional interfaces

Q7. What does the @FunctionalInterface do, and how does it differ from the @Override annotation?
A7. The @Override annotation takes advantage of the compiler checking to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does. Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.

The annotation **@FunctionalInterface** acts similarly to @Override by signalling to the compiler that the interface is intended to be a functional interface. The compiler will then throw a **compile-time** error

1) If the interface has no abstract methods. Try removing the “sum” method shown above, you will get the following compile-time error: “Invalid ‘@FunctionalInterface’ annotation; Summable is not a functional interface”.

2) If the interface has multiple abstract methods. Following gives a compile-time error.

```

@FunctionalInterface
public interface Summable {
    public int sum(int input1, int input2);
    public int subtract(int input1, int input2);
}

```

The “@FunctionalInterface” annotation is an **optional** facility to avoid **accidental addition of abstract methods** in the functional interfaces. It is a good practice to use this annotation where applicable.

Q8. Can you have method implantations in Java interfaces?
A8. Yes. Prior to Java 8, you could have only “method

declarations" in the interfaces. One of the key design changes in Java 8 is that you can have **default methods** and **static methods** in the interfaces. This means, from Java 8 onwards you can define behaviors (i.e. implementation) in interfaces.

Q9. Does this mean that abstract classes are not required any more?
A9. No. You can only define behaviors via default and static methods, but you can't maintain states by defining non-final variables. Absract classes are required to define default behavior with state.

Q10. Can a functional interface have any number of default and static methods?
A10. Yes. only 1 abstract method (e.g sum), and any number of default and static methods as shown below.

```java


@FunctionalInterface
public interface Summable {

    static final int var = 0;

    int sum(int input1, int input2);

    // static method
    static int subtract(int input1, int input2)
        return input1 - input2;
    }

    // default method
    default int multiply(int input1, int input2)
        return input1 * input2;
    }

    // default method
    default int divide(int input1, int input2) {
        return input1 / input2;
    }
}

```

Q11. Why were default methods introduced in Java 8?
A11. In order to support functional programming, new intermediate and terminal operations like map, forEach, etc had to be added so that the Java collection API can work with lambda expressions like "i -> i * i" and "i -> {System.out.print(i

+ " ");}". One way to enable this is by adding these new methods to existing interfaces like java.util.Collection, java.util.List, etc and then providing the implementations where required. But this approach has a problem. Once the JDK is published, it would not be possible to add new methods to those interfaces by extending those interfaces without breaking the existing implementation. Hence this new concept of "default methods" was introduced to provide default implementation of the declared behavior.

An example where a collection if integers is converted to a stream and then printed

```

List<Integer> numbers = Arrays.asList(1,2,3,4,5);
numbers.stream().forEach(i -> System.out.print(i

```

The stream() method returns a "**Stream**" interface, which has all the methods that take **functional interfaces** like Supplier, BiConsumer, Function, etc as arguments.

```

package java.util.stream;

public interface Stream<T> extends BaseStream<T,

    <R> Stream<R> map(Function<? super T, ? exte

    void forEach(Consumer<? super T> action);

    <R> R collect(Supplier<R> supplier,
                  BiConsumer<R, ? super T> accum
                  BiConsumer<R, R> combiner);


    //more abstract and static methods
}

```

Q12. What are the benefits of default and static methods?
A12.

1) Default methods enhance the Collection API to support lambda expressions.

2) Default methods will help you in extending existing interfaces without breaking the implementation classes as explained in "A11".

3) Default methods eliminate the need for the base classes to provide default behaviors. The interfaces can provide default behaviors, and the implementing classes can choose to override the default behaviors.

4) Default and Static methods can be added to the interface itself without requiring separate utility classes like java.util.Collections, java.util.Arrays, etc.

5) The interface static methods are handy for providing utility methods like null checks and collection sorting/shuffling.

6) Unlike the default methods, the interface static methods prevent the implementing classes from accidentally overriding them.

Q13. What happens to a class that implements two or more interfaces with the same default method name and signature?
A13. You will get a compile-time error in the implementing class to define its own method with the same default method name and signature to resolve the conflict.

Q14. Can a default method override a method from java.lang.Object class?
A14. No. It's not possible because the "java.lang.Object" is the implicit base class for all Java classes. This means even if you have Object class methods defined as default methods in interfaces, the Object class method will always be used. So, no point in allowing interfaces to override a method from java.lang.Object class.

Q15. Can an interface static method override a method from java.lang.Object class?
A15. No. You will get a compile-time error: "This static method cannot hide the instance method from Object". As mentioned earlier, the Object is implicitly the base class for all

the other Java objects and you can't have one class level static method and another instance level method with the same signature.

Q16. What is currying? Does Java 8 support currying?
A16. 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.

Learn more with examples: What is currying? Does Java 8 support currying?

Q17. What do you understand by the term referential transparency with respect to functional programming?
A17. In FP, a function will return the same result for invocations with the same parameters, and this is known as "referential transparency". This allows the result to be easily cached and returned any number of times to improve performance. This enables **lazy evaluation** that I mentioned earlier to defer the computation of values until the point when they are needed.

Q18. What is a pure function?
A18. Functions with absolutely **no side effects** or functions that operate on immutable data are known as pure functions. This has several benefits. Firstly, since the data can not be changed accidentally or on purpose, it can be freely shared improving memory requirements and enabling parallelism.

Q19. What is a closure?
A19. A closure is function that can be stored as a variable, and passed around to other functions with a special ability to access other variables local to the scope it was created in. Since functions are treated like first class citizens, it's really useful to be able to pass them around together with their referencing environment (i.e. a reference to each non-local variable of that function).

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.

‹ ♦ 11 Spring boot interview questions & answers

♥ How to turn readers of your Java CV go from "Blah blah" to "Wow"? ›

Posted in FP, Java 8, member-paid, OOP & FP Essentials

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