(/)

Java 8 Interview Questions(+ Answers)

Last modified: September 10, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) +

Interview (https://www.baeldung.com/tag/interview/) **Java 8 (https://www.baeldung.com/tag/java-8/)**

This article is part of a series:

1. Introduction

In this article, we are going to explore some of the JDK8-related questions that might pop up during an interview.

Java 8 is a platform release packed with new language features and library classes. Most of these new features are geared towards achieving cleaner and more compact code, and some add new functionality that has never been supported in Java.

2. Java 8 General Knowledge

Q1. What new features were added in Java 8?

Java 8 ships with several new features but the most significant are the following:

- **Lambda Expressions** – a new language feature allowing treating actions as objects
- **Method References** – enable defining Lambda Expressions by referring to methods directly using their names
- ***Optional*** – special wrapper class used for expressing optionality
- **Functional Interface** – an interface with maximum one abstract method, implementation can be provided using a Lambda Expression
- **Default methods** – give us the ability to add full implementations in interfaces besides abstract methods
- **Nashorn, JavaScript Engine** – Java-based engine for executing and evaluating JavaScript code
- ***Stream*** **API** – a special iterator class that allows processing collections of objects in a functional manner
- **Date API** – an improved, immutable JodaTime-inspired Date API

Along with these new features, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

3. Method References

Q1. What is a method reference?

A method reference is a Java 8 construct that can be used for referencing a method without invoking it. It is used for treating methods as Lambda Expressions. They only work as syntactic sugar to reduce the verbosity of some lambdas. This way, the following code:

```
1 (o) -> o.toString();
```

can become:

```
1 Object::toString();
```

A method reference can be identified by a double colon separating a class or object name and the name of the method. It has different variations such as constructor reference:

```
1 String::new;
```

Static method reference:

```
1 String::valueOf;
```

Bound instance method reference:

```
str::toString;
```

Unbound instance method reference:

```
String::toString;
```

You can read a detailed description of method references with full examples by following this link (https://www.codementor.io/eh3rrera/using-java-8-method-reference-du10866vx) and this one (/java-8-double-colon-operator).

Q2. What is the meaning of String::valueOf expression?

It is a static method reference to the *valueOf* method of the *String* class.

4. Optional

Q1. What is *Optional*? How can it be used?

Optional is a new class in Java 8 that encapsulates an optional value i.e. a value that is either there or not. It is a wrapper around an object, and you can think of it as a container of zero or one element.

Optional has a special *Optional.empty()* value instead of wrapped *null*. Thus it can be used instead of a nullable value to get rid of *NullPointerException* in many cases.

You can read a dedicated article about *Optional* here (/java-optional).

The main purpose of *Optional*, as designed by its creators, was to be a return type of methods that previously would return *null*. Such methods would require you to write boilerplate code to check the return value and sometimes could forget to do a defensive check. In Java 8, an *Optional* return type explicitly requires you to handle null or non-null wrapped values differently.

For instance, the *Stream.min()* method calculates the minimum value in a stream of values. But what if the stream is empty? If it was not for *Optional*, the method would return *null* or throw an exception.

But it returns an *Optional* value which may be *Optional.empty()* (the second case). This allows us to easily handle such case:

```
int min1 = Arrays.stream(new int[]{1, 2, 3, 4, 5})
  .min()
  .orElse(0);
assertEquals(1, min1);

int min2 = Arrays.stream(new int[]{})
  .min()
  .orElse(0);
assertEquals(0, min2);
```

It's worth noting that *Optional* is not a general purpose class like *Option* in Scala. It is not recommended to be used as a field value in entity classes, which is clearly indicated by it not implementing the *Serializable* interface.

5. Functional Interfaces

Q1. Describe some of the functional interfaces in the standard library.

There are a lot of functional interfaces in the *java.util.function* package, the more common ones include but not limited to:

- *Function* – it takes one argument and returns a result
- *Consumer* – it takes one argument and returns no result (represents a side effect)
- *Supplier* – it takes not argument and returns a result
- *Predicate* – it takes one argument and returns a boolean
- *BiFunction* – it takes two arguments and returns a result
- *BinaryOperator* – it is similar to a *BiFunction*, taking two arguments and returning a result. The two arguments and the result are all of the same types
- *UnaryOperator* – it is similar to a *Function*, taking a single argument and returning a result of the same type

For more on functional interfaces, see the article "Functional Interfaces in Java 8" (/java-8-functional-interfaces).

Q2. What is a functional interface? What are the rules of defining a functional interface?

A functional interface is an interface with no more, no less but one single abstract method (*default* methods do not count).

Where an instance of such interface is required, a Lambda Expression can be used instead. More formally put: *Functional interfaces* provide target types for lambda expressions and method references.

The arguments and return type of such expression directly match those of the single abstract method.

For instance, the *Runnable* interface is a functional interface, so instead of:

```
Thread thread = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
});
```

you could simply do:

```
Thread thread = new Thread(() -> System.out.println("Hello World!"));
```

Functional interfaces are usually annotated with the *@FunctionalInterface* annotation – which is informative and does not affect the semantics.

6. Default Method

Q1. What is a default method and when do we use it?

A default method is a method with an implementation – which can be found in an interface.

We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface:

```
public interface Vehicle {
    public void move();
    default void hoot() {
        System.out.println("peep!");
    }
}
```

Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method. In Java 8, this problem has been solved by the use of default method.

For example, *Collection* interface does not have *forEach* method declaration. Thus, adding such method would simply break the whole collections API.

Java 8 introduces default method so that *Collection* interface can have a default implementation of *forEach* method without requiring the classes implementing this interface to implement the same.

Q2. Will the following code compile?

```
@FunctionalInterface
public interface Function2<T, U, V> {
    public V apply(T t, U u);

    default void count() {
        // increment counter
    }
}
```

Yes. The code will compile because it follows the functional interface specification of defining only a single abstract method. The second method, *count*, is a default method that does not increase the abstract method count.

7. Lambda Expressions

Q1. What is a Lambda Expression and what is it used for

In very simple terms, a lambda expression is a function that can be referenced and passed around as an object.

Lambda expressions introduce functional style processing in Java and facilitate the writing of compact and easy-to-read code.

Because of this, lambda expressions are a natural replacement for anonymous classes as method arguments. One of their main uses is to define inline implementations of functional interfaces.

Q2. Explain the syntax and characteristics of a Lambda Expression

A lambda expression consists of two parts: the parameter part and the expressions part separated by a forward arrow as below:

```
1 params -> expressions
```

Any lambda expression has the following characteristics:

- **Optional type declaration** – when declaring the parameters on the left-hand side of the lambda, we don't need to declare their types as the compiler can infer them from their values. So *int param -> …* and *param ->…* are all valid
- **Optional parentheses** – when only a single parameter is declared, we don't need to place it in parentheses. This means *param -> …* and *(param) -> …* are all valid. But when more than one parameter is declared, parentheses are required
- **Optional curly braces** – when the expressions part only has a single statement, there is no need for curly braces. This means that *param – > statement* and *param – > {statement;}* are all valid. But curly braces are required when there is more than one statement
- **Optional return statement** – when the expression returns a value and it is wrapped inside curly braces, then we don't need a return statement. That means *(a, b) – > {return a+b;}* and *(a, b) – > {a+b;}* are both valid

To read more about Lambda expressions, follow this link (https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm) and this one (/java-8-lambda-expressions-tips).

8. Nashorn Javascript

Q1. What is Nashorn in Java8?

Nashorn (/java-nashorn) is the new Javascript processing engine for the Java platform that shipped with Java 8. Until JDK 7, the Java platform used Mozilla Rhino for the same purpose. as a Javascript processing engine.

Nashorn provides better compliance with the ECMA normalized JavaScript specification and better runtime performance than its predecessor.

Q2. What is jjs?

In Java 8, *jjs* is the new executable or command line tool used to execute Javascript code at the console.

9. Streams

Q1. What is a stream? How does it differ from a collection?

In simple terms, a stream is an iterator whose role is to accept a set of actions to apply on each of the elements it contains.

The stream represents a sequence of objects from a source such as a collection, which supports aggregate operations. They were designed to make collection processing simple and concise. Contrary to the collections, the logic of iteration is implemented inside the stream, so we can use methods like *map* and *flatMap* for performing a declarative processing.

Another difference is that the *Stream* API is fluent and allows pipelining:

```
int sum = Arrays.stream(new int[]{1, 2, 3})
  .filter(i -> i >= 2)
  .map(i -> i * 3)
  .sum();
```

And yet another important distinction from collections is that streams are inherently lazily loaded and processed.

Q2. What is the difference between intermediate and terminal operations?

Stream operations are combined into pipelines to process streams. All operations are either intermediate or terminal.

Intermediate operations are those operations that return *Stream* itself allowing for further operations on a stream.

These operations are always lazy, i.e. they do not process the stream at the call site, an intermediate operation can only process data when there is a terminal operation. Some of the intermediate operations are *filter*, *map* and *flatMap*.

Terminal operations terminate the pipeline and initiate stream processing. The stream is passed through all intermediate operations during terminal operation call. Terminal operations include *forEach*, *reduce, Collect* and *sum*.

To drive this point home, let us look at an example with side effects:

```
public static void main(String[] args) {
    System.out.println("Stream without terminal operation");

    Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {
        System.out.println("doubling " + i);
        return i * 2;
    });

    System.out.println("Stream with terminal operation");
        Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {
            System.out.println("doubling " + i);
            return i * 2;
    }).sum();
}
```

The output will be as follows:

```
Stream without terminal operation
Stream with terminal operation
doubling 1
doubling 2
doubling 3
```

As you can see, the intermediate operations are only triggered when a terminal operation exists.

Q3. What is the difference between *map* and *flatMap* stream operation?

There is a difference in signature between *map* and *flatMap*. Generally speaking, a *map* operation wraps its return value inside its ordinal type while *flatMap* does not.

For example, in *Optional*, a *map* operation would return *Optional<String>* type while *flatMap* would return *String* type.

So after mapping, one needs to unwrap (read "flatten") the object to retrieve the value whereas, after flat mapping, there is no such need as the object is already flattened. The same concept is applied to mapping and flat mapping in *Stream*.

Both *map* and *flatMap* are intermediate stream operations that receive a function and apply this function to all elements of a stream.

The difference is that for the *map*, this function returns a value, but for *flatMap*, this function returns a stream. The *flatMap* operation "flattens" the streams into one.

Here's an example where we take a map of users' names and lists of phones and "flatten" it down to a list of phones of all the users using *flatMap*:

```
Map<String, List<String>> people = new HashMap<>();
people.put("John", Arrays.asList("555-1123", "555-3389"));
people.put("Mary", Arrays.asList("555-2243", "555-5264"));
people.put("Steve", Arrays.asList("555-6654", "555-3242"));

List<String> phones = people.values().stream()
  .flatMap(Collection::stream)
    .collect(Collectors.toList());
```

Q4. What is stream pipelining in Java 8?

Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories: intermediate operations and terminal operations.

Each intermediate operation returns an instance of Stream itself when it runs, an arbitrary number of intermediate operations can, therefore, be set up to process data forming a processing pipeline.

There must then be a terminal operation which returns a final value and terminates the pipeline.

10. Java 8 Date and Time API

Q1. Tell us about the new Date and Time API in Java 8

A long-standing problem for Java developers has been the inadequate support for the date and time manipulations required by ordinary developers.

The existing classes such as *java.util.Date* and *SimpleDateFormatter* aren't thread-safe, leading to potential concurrency issues for users.

Poor API design is also a reality in the old Java Data API. Here's just a quick example – years in *java.util.Date* start at 1900, months start at 1, and days start at 0 which is not very intuitive.

These issues and several others have led to the popularity of third-party date and time libraries, such as Joda-Time.

In order to address these problems and provide better support in JDK, a new date and time API, which is free of these problems, has been designed for Java SE 8 under the package *java.time*.

11. Conclusion

In this article, we've explored a few very important questions for technical interview questions with a bias on Java 8. This is by no means an exhaustive list but only contains questions that we think are most likely to appear in each new feature of Java 8.

Even if you are just starting up, ignorance of Java 8 isn't a good way to go in an interview, especially when Java appears strongly on your resume. It is, therefore, important that you take some time to understand the answers to these questions and possibly do more research.

Good luck in your interview.

Next »

Memory Management in Java Interview Questions (+Answers) (https://www.baeldung.com/java-memory-management-interview-questions)

« Previous

Java Class Structure and Initialization Interview Questions (https://www.baeldung.com/java-classes-initialization-questions)

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)