Java 8 Interview Questions and Answers

Author: Ramesh Fadatare
Interview Java 8
In this article, we will discuss some important and frequently asked Java 8 Interview Questions and Answers.

Learn everything about Java 8 at Java 8 Tutorial

YouTube Video

1. What new features were added in Java 8?

In Java 8, the following new features were added:

- Lambda Expressions − lambda expression is a function that can be referenced and passed around as an object
- Method References − Method references are the references that use a function as a parameter to request a method.
- Optional − This class is to provide a type-level solution for representing optional values instead of using null references.
- Functional Interface – An Interface that contains exactly one abstract method and implementation can be provided using a Lambda Expression
- Default methods − give us the ability to add full implementations in interfaces besides abstract methods
- Stream API − Stream API provides a functional approach to processing collections of objects.
- Date and Time API − an improved, immutable JodaTime-inspired Date API
- Nashorn, JavaScript Engine − Java-based engine for executing and evaluating JavaScript code

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

The below diagram shows all the Java 8 features and enhancements.

2. What is a Lambda Expression?

The lambda expression is simply a function without any name. It can even be used as a parameter in a function. Lambda Expression facilitates functional programming and simplifies development a lot.

The main use of Lambda expression is to provide an implementation for **functional interfaces.**

For example, Lambda expression provides an implementation for a *Printable* functional interface

```
interface Printable {
    void print(String msg);
}

public class JLEExampleSingleParameter {

    public static void main(String[] args) {
         // without lambda expression
         Printable printable = new Printable() {
            @Override
            public void print(String msg) {
               System.out.println(msg);
            }
         };
         printable.print(" Print message to console....");

         // with lambda expression
         Printable withLambda = (msg) -> System.out.println(msg);
         withLambda.print(" Print message to console....");
     }
}
```

Output :

```
 Print message to console....
 Print message to console....
```

Example 2, Create a method that takes a lambda expression as a parameter:

```
interface StringFunction {
    String run(String str);
}

public class Main {
     public static void main(String[] args) {
       StringFunction exclaim = (s) -> s + "!";
       StringFunction ask = (s) -> s + "?";
       printFormatted("Hello", exclaim);
       printFormatted("Hello", ask);
    }
    public static void printFormatted(String str, StringFunction format) {
       String result = format.run(str);
       System.out.println(result);
     }
}
```

Example 3, Pass lambda expression as an argument to the constructor

```
    static Runnable runnableLambda = () -> {
        System.out.println("Runnable Task 1");
        System.out.println("Runnable Task 2");
    };
   //Pass lambda expression as argument
    new Thread(runnableLambda).start();
```

Read more in detail about lambda expressions at Java 8 Lambda Expressions

3. Why use Lambda Expression?

1. **Facilitates functional programming** - Lambda Expression facilitates functional programming and simplifies the development a lot.
2. To provide the implementation of the Java 8 Functional Interface.
3. **Reduced Lines of Code** - One of the clear benefits of using lambda expression is that the amount of code is reduced, we have already seen how easily we can create instances of a functional interface using lambda expression rather than using an anonymous class.
4. **Passing Behaviors into methods** - Lambda Expressions enable you to encapsulate a single unit of behavior and pass it to other code. For example, to other methods or constructors.

Read more in detail about lambda expressions at Java 8 Lambda Expressions.

4. Explain Lambda Expression Syntax

Java Lambda Expression Syntax:

```
(argument-list) -> {body}
```

Java lambda expression consists of three components.

- **Argument list:** It can be empty or non-empty as well.
- **Arrow-token:** It is used to link arguments list and body of expression.
- **Body:** It contains expressions and statements for the lambda expression.

For example, Consider we have a functional interface:

```
interface Addable{
    int add(int a,int b);
}
```

Let's implement the above *Addable* functional interface using a lambda expression:

```
        Addable withLambdaD = (int a,int b) -> (a+b);
        System.out.println(withLambdaD.add(100,200));
```

Read more in detail about lambda expressions at Java 8 Lambda Expressions.

5. What is a functional interface?

An Interface that contains exactly one abstract method is known as a **functional interface**. It can have any number of default, static methods but can contain only one abstract method. It can also declare the methods of the object class.

Functional Interface is also known as **Single Abstract Method** Interfaces or SAM Interfaces. A functional interface can extend another interface only when it does not have any abstract method.

Java 8 provides predefined functional interfaces to deal with functional programming by using lambda and method references.

For example:

```
interface Printable {
    void print(String msg);
}

public class JLEExampleSingleParameter {

    public static void main(String[] args) {
         // with lambda expression
         Printable withLambda = (msg) -> System.out.println(msg);
         withLambda.print(" Print message to console....");
     }
}
```

Output :

```
 Print message to console....
```

Read more at Java 8 Functional Interfaces with Examples.

6. Is it possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?

Yes, it is possible to define our own Functional Interfaces. We use Java 8 to provide the *@FunctionalInterface* annotation to mark an interface as a Functional Interface.

We need to follow these rules to define a Functional Interface:

- Define an interface with one and only one abstract method.
- We cannot define more than one abstract method.
- Use *@FunctionalInterface* annotation in the interface definition.
- We can define any number of other methods like default methods, static methods.

The below example illustrates defining our own Functional Interface:

Let's create a *Sayable* interface annotated with *@FunctionalInterface* annotation.

```
@FunctionalInterface
interface Sayable{
    void say(String msg);    // abstract method
}
```

Let's demonstrate a custom functional interface via the *main()* method.

```
public class FunctionalInterfacesExample {

    public static void main(String[] args) {

        Sayable sayable = (msg) -> {
            System.out.println(msg);
        };
        sayable.say("Say something ..");
    }
}
```

Read more at Java 8 Functional Interfaces with Examples.

7. Name some of the functional interfaces in the standard library

In Java 8, there are a lot of functional interfaces introduced in the *java.util.function* package and the more common ones include but are 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 no argument and returns a result
- *Predicate* – it takes one argument and returns a boolean
- *BiFunction* – it takes two arguments and returns a result
- BiConsumer - it takes two (reference type) input arguments and returns no 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

- *Runnable*: use to execute the instances of a class over another thread with no arguments and no return value.
- *Callable*: use to execute the instances of a class over another thread with no arguments and it either returns a value or throws an exception.
- *Comparator*: use to sort different objects in a user-defined order
- *Comparable*: use to sort objects in the natural sort order

> For more on functional interfaces, see the article at Java 8 Functional Interfaces with Examples.

8. What is a method reference?

Method reference is used to refer method of the functional interface. It is a compact and easy form of lambda expression. Each time when you are using a lambda expression to just referring a method, you can replace your lambda expression with a method reference.

Below are a few examples of method references:

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

can become:

```
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:

```
String::new;
```

Static method reference:

```
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 at Java 8 Method References.

9. What are different kinds of Method References?

There are four kinds of method references:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

1. Reference to a static method. For example:

```
ContainingClass::staticMethodName
```

2. Reference to an instance method of a particular object. For example:

```
containingObject::instanceMethodName
```

3. Reference to an instance method of an arbitrary object of a particular type. For example:

```
ContainingType::methodName
```

4. Reference to a constructor. for example:

```
ClassName::new
```

You can read a detailed description of method references with full examples at Java 8 Method References.

10. What is a Stream? How to create Streams in Java?

A stream in Java represents a sequence of elements and supports different kinds of operations to perform computations upon those elements. Streams are a core part of Java's functional programming paradigm, introduced in Java 8.

Streams can be obtained in various ways from different sources such as collections, arrays, I/O channels, etc. They don't change the original data structure; they just provide a view of this data structure to allow bulk operations.

Key Characteristics of Streams:

No Storage: Streams have no storage for elements; they just convey elements from a source through a pipeline of computational operations.

Functional in Nature: An operation on a stream produces a result, but it does not modify the source. It returns a new stream that contains the result.

Laziness-Seeking: Many stream operations are "lazy," meaning that they do not process elements until the result is needed.

Possibly Unbounded: Streams can represent sequences that are of an infinite size.

Consumable: The elements of a stream are consumed from the data source only when queried. Once traversed, they are not revisited.

Creating Streams in Java

From a Collection:

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
Stream<String> myStream = myList.stream();
```

From an Array:

```
int[] numbers = {1, 2, 3, 4, 5};
IntStream numberStream = Arrays.stream(numbers);
```

Using Stream.of():

```
Stream<String> stringStream = Stream.of("A", "B", "C");
```

Using Stream.iterate() for Infinite Streams:

```
Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);
```

Read more about streams at Java 8 Stream APIs with Examples.

11. Collection API vs Stream API?

Collection API	Stream API
It's available since Java 1.2	It is introduced in Java SE 8
It is used to store Data (A set of Objects).	It is used to compute data (Computation on a set of Objects).
We can use both *Spliterator* and *Iterator* to iterate elements.	We can't use *Spliterator* or *Iterator* to iterate elements. We can use *forEach* to perform an action for each element of this stream.
It is used to store an unlimited number of elements.	Stream API is used to process the elements of a Collection.
Typically, it uses the External Iteration concept to iterate Elements such as Iterators.	Stream API uses internal iteration to iterate Elements, using the *forEach* method.
Collection Object is constructed Eagerly.	Stream Object is constructed Lazily.

Collection API	Stream API
We add elements to the Collection object only after it is computed completely.	We can add elements to Stream Object without any prior computation. That means Stream objects are computed on demand.

12. What is Optional in Java 8?

Optional is a container object which is used to contain not-null objects. An optional object is used to represent null with an absent value.

The Optional class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values.

The purpose of the Optional class is to provide a type-level solution for representing optional values instead of using null references.

Read more about Optional Class with examples at Java 8 Optional Class with Examples.

13. What are the Advantages of Java 8 Optional?

- Null checks are not required.
- No more **NullPointerException** at run-time.
- We can develop clean and neat APIs.
- No more Boilerplate code

Read more about Optional Class with examples at Java 8 Optional Class with Examples.

14. 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 new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface:

```
public interface Vehicle {
    String getBrand();

    String speedUp();
```

```
    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehice alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

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 the default method.

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

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

Read more about Default Methods with examples at Java 8 Static and Default Methods in Interface.

15. What is a Static Method? Why do we need Static methods in Java 8 Interfaces?

A Static Method is a Utility method or Helper method, which is associated with a class (or interface). It is not associated with any object.
We need Static Methods because of the following reasons:

- We can keep Helper or Utility methods specific to an interface in the same interface rather than in a separate Utility class.
- We do not need separate Utility Classes like Collections, Arrays, etc to keep Utility methods.
- Clear separation of Responsibilities. That is we do not need one Utility class to keep all Utility methods of Collection API like Collections etc.
- Easy to extend the API.
- Easy to Maintain the API.

16. How will you call a default method of an interface in a class?

Using the *super* keyword along with the interface name.

```
interface Vehicle {
   default void print() {
      System.out.println("I am a vehicle!");
   }
}
class Car implements Vehicle {
   public void print() {
      Vehicle.super.print();
   }
}
```

17. How will you call a static method of an interface in a class?

Using the name of the interface.

```
interface Vehicle {
   static void blowHorn() {
      System.out.println("Blowing horn!!!");
   }
}
class Car implements Vehicle {
   public void print() {
      Vehicle.blowHorn();
   }
}
```