Blog Site: Java Guides Author: Ramesh Fadatare

Java 8 Interview Questions and Answers

In this article, we will discuss some important Java 8 Interview Questions and Answers.

I would like to share my experience with Java 8 interview questions. Let's list all commonly asked interview questions and answers regarding Java 8 features and enhancement.

1. What new features were added in Java 8?
2. What is a Lambda Expression?
3. Why use Lambda Expression?
4. Explain Lambda Expression Syntax
5. What is a functional interface?
6. Is it possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?
7. Describe some of the functional interfaces in the standard library.
8. What is a functional interface? What are the rules of defining a functional interface?
9. What is a method reference?
10. What are different kinds of Method References?
11. What is a streams in Java 8 ? How does it differ from a collection?
12. What is stream pipelining in Java 8?
13. Explain Differences between Collection API and Stream API?
14. What is Optional in Java 8?
15. What are Advantages of Java 8 Optional?
16. What are Collectors Class in Java 8?
17. What is use of Java 8 StringJoiner Class?
18. What is a default method and when do we use it?
19. Why Default Methods in Interfaces Are Needed?
20 .What is use of static methods in Interface?
21. How will you call a default method of an interface in a class?
22. How will you call a static method of an interface in a class?
23.What is Diamond Problem in Inheritance? How Java 8 Solves this problem?
24. What is Nashorn in Java8?

26. What is jjs?
27. What are Java 8 Date-Time API Benefits over Old Date and Calender Classes?

Note that this article not only helps you to prepare for interviews but also help you to refresh your Java 8 knowledge with simple definition and examples.

1. What new features were added in Java 8?

Java 8 ships with several new features and enhancements 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** – This class is to provide a type-level solution for representing optional values instead of using null references.
- **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 and Time 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.

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

Blog Site: Java Guides Author: Ramesh Fadatare

2. What is a Lambda Expression?

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

3. Why use Lambda Expression?

1.To provide the implementation of the **Java 8 Functional Interface**.
2.Less coding - lambda expressions are a natural replacement for anonymous classes as a method argument
3.Lambda Expressions enable you to encapsulate a single unit of behavior and pass it to other code. You can use lambda expressions if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.

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.

The below diagram shows the important characteristics of a lambda expression.

Java Lambda Expression Syntax Cheat Sheet

Generic Java Lambda Expression Syntax: `(argument-list) -> {body}`

Java Lambda Expression with No Parameter: `() -> { System.out.println("Lambda Expression"); };`

Java Lambda Expression with Single Parameter: `(msg) -> System.out.println(msg); }`

Java Lambda Expression with Multiple Parameters: `(int a,int b) -> (a+b);`

Java Lambda Expression without return keyword: `(a, b) -> (a + b);`

Java Lambda Expression with Multiple Statements:

```
(x,y) -> {
   System.out.println("x : " + x);
   System.out.println("y: " + y);
   return (x+y);
};
```

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 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, below Java program to illustrate *`Predicate`* functional interface usage.

```
// Java program to illustrate Simple Predicate

import java.util.function.Predicate;
public class PredicateInterfaceExample {
    public static void main(String[] args)
    {
        // Creating predicate
        Predicate<Integer> lesserthan = i -> (i < 18);
```

```
        // Calling Predicate method
        System.out.println(lesserthan.test(10));
    }
}
```

Output

```
true
```

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 SE 8's *@FunctionalInterface* annotation to mark an interface as 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.
- If we override java.lang.Object class's method as an abstract method, which does not count as an abstract method.

Below example illustrate the defining our own Functional Interface:
Let's create *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. 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 at **Java 8 Functional Interfaces with Examples**.

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

An Interface that contains exactly one abstract method is known 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.
- If we override `java.lang.Object` class's method as an abstract method, which does not count as an abstract method.

9. What is a method reference?

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

```
(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**.

10. 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**.

11. What is a streams in Java 8 ? 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.

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

12. 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.

The following example prints the male members contained in the collection roster with a pipeline that consists of the aggregate operations filter and *forEach*:

```
roster
```

```
        .stream()
        .filter(e -> e.getGender() == Person.Sex.MALE)
        .forEach(e -> System.out.println(e.getName()));
```

Read more at **Collection Aggregate Operations using Streams**

13. Explain Differences between Collection API and Stream API?

S.NO.	COLLECTION API	STREAM API
1.	It's available since Java 1.2	It is introduced in Java SE8
2.	It is used to store Data(A set of Objects).	It is used to compute data(Computation on a set of Objects).
3.	We can use both Spliterator and Iterator to iterate elements. We can use forEach to performs an action for each element of this stream.	We can't use Spliterator or Iterator to iterate elements.
4.	It is used to store limited number of Elements.	It is used to store either Limited or Infinite Number of Elements.
5.	Typically, it uses External Iteration concept to iterate Elements such as Iterator.	Stream API uses External Iteration to iterate Elements, using forEach methods.
6.	Collection Object is constructed Eagerly.	Stream Object is constructed Lazily.
7.	We add elements to 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.
8.	We can iterate and consume elements from a Collection Object at any number of times.	We can iterate and consume elements from a Stream Object only once.

14. What is Optional in Java 8?

Optional is a container object which is used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values. It is introduced in **Java 8** and is similar to what Optional is in Guava.
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**.

15. What are Advantages of Java 8 Optional?

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

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

16. What are Collectors Class in Java 8?

Collectors is a final class that extends *Object* class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.
The following are examples of using the predefined collectors to perform common mutable reduction tasks:

```
        // Accumulate names into a List
        List<String> list =
people.stream().map(Person::getName).collect(Collectors.toList());

        // Accumulate names into a TreeSet
        Set<String> set =
people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));

        // Convert elements to strings and concatenate them, separated by commas
        String joined = things.stream()
```

```
                                .map(Object::toString)
                                .collect(Collectors.joining(", "));

        // Compute sum of salaries of employee
        int total = employees.stream()
                                .collect(Collectors.summingInt(Employee::getSalary)));

        // Group employees by department
        Map<Department, List<Employee>> byDept
             = employees.stream()
                        .collect(Collectors.groupingBy(Employee::getDepartment));

        // Compute sum of salaries by department
        Map<Department, Integer> totalByDept
             = employees.stream()
                        .collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.summingInt(Employee::getSalary)));

        // Partition students into passing and failing
        Map<Boolean, List<Student>> passingFailing =
             students.stream()
                      .collect(Collectors.partitioningBy(s -> s.getGrade() >=
PASS_THRESHOLD));
```

Read more about Collectors class at **Java 8 Collectors Class with Examples**.

17. What is use of Java 8 StringJoiner Class?

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, we can create a string by passing delimiters like a comma(,), hyphen(-) etc.

Example: Simple Delimiters Example

```
 private static void delimiterDemonstration() {
  StringJoiner joinNames = new StringJoiner(","); // passing comma(,) as delimiter
```

```
    // Adding values to StringJoiner
    joinNames.add("Rahul");
    joinNames.add("Raju");
    joinNames.add("Peter");
    joinNames.add("Raheem");
    System.out.println(joinNames);

    joinNames = new StringJoiner("|"); // passing comma(,) as delimiter

    // Adding values to StringJoiner
    joinNames.add("Rahul");
    joinNames.add("Raju");
    joinNames.add("Peter");
    joinNames.add("Raheem");
    System.out.println(joinNames);
}
```

Read more about *StringJoiner* class at **Java 8 StringJoiner Class with Examples**.

18. 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 {
     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 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.
Read more about Default Methods with examples at **Java 8 Static and Default Methods in Interface**.

19. Why Default Methods in Interfaces Are Needed?

•Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

•In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down so default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Thus, there's no need to modify the implementing classes. In this way, backward compatibility is neatly preserved without having to refactor the implementers.

•The default method is used to define a method with a default implementation. You can override the default method also to provide the more specific implementation for the method.

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

20 .What is use of static methods in Interface?

The idea behind static interface methods is to provide a simple mechanism that allows us to increase the degree of cohesion of a design by putting together related methods in one single place without having to create an object.
Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.
For example:

```
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.";
     }

     static int getHorsePower(int rpm, int torque) {
          return (rpm * torque) / 5252;
     }
}
```

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

Using *super* keyword along with interface name.

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

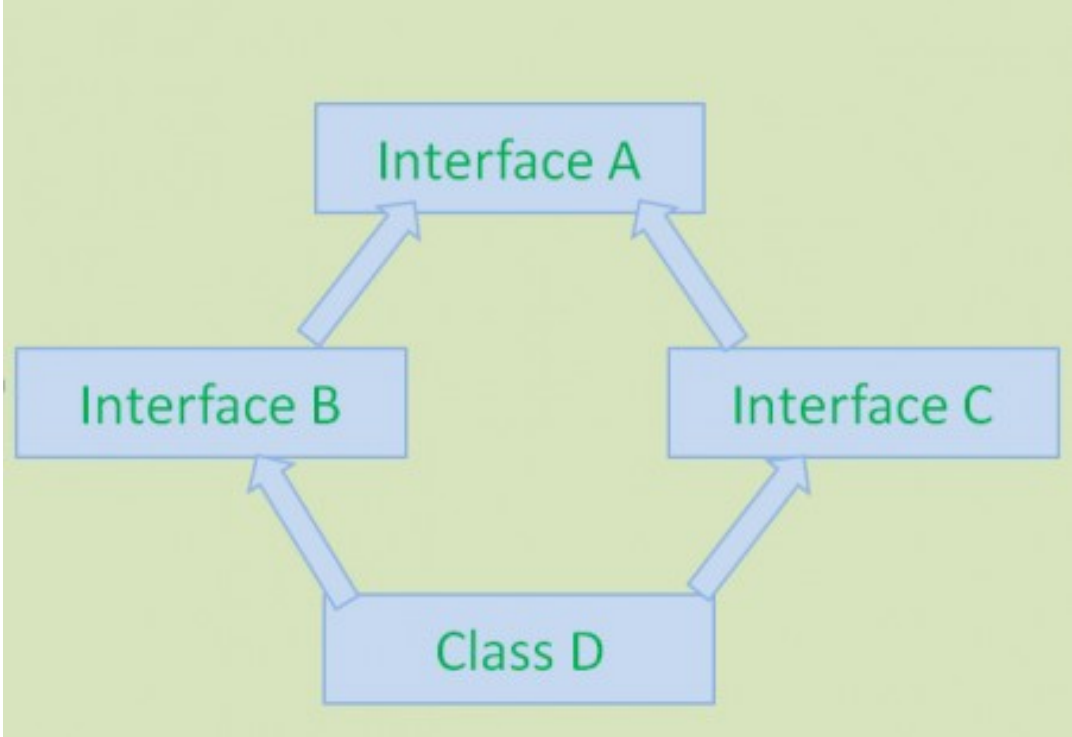
22. How will you call a static method of an interface in a class?

Using name of the interface.

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

23.What is Diamond Problem in Inheritance? How Java 8 Solves this problem?

A Diamond Problem is a Multiple Inheritance problem. In Java, It occurs when a Class extends more than one Interface which have same method implementation (Default method).



This above diagram shows Diamond Problem. To avoid this problem, **Java 7** and Earlier versions does not support methods implementation in interface and also doesn't support Multiple Inheritance. Java 8 has introduced new feature: Default methods to support Multiple Inheritance with some limitations.

Sample Java SE 8 Code to show this Diamond Problem:

```
public interface A{
  default void display() { //code goes here }
}
public interface B extends A{ }
public interface C extends A{ }
public class D implements B,C{ }
```

In the above code snippet, class D gives compile time errors because Java Compiler will get bit confusion about which `display()` has to provide in class D. Class D inherits `display()` method from both interfaces B and C. To solve this problem, Java SE 8 has given the following remedy:

```
public interface A{
  default void display() { //code goes here }
}
public interface B extends A{ }
public interface C extends A{ }
public class D implements B,C{
  void display() {
    B.super.display();
  }
}
```

This `B.super.display();` will solve this Diamond Problem.

24. What is Nashorn in Java8?

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.

26. What is jjs?

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

27. What are Java 8 Date-Time API Benefits over Old Date and Calender Classes?

Here are some of the challenges faced by developers with earlier Java versions:

1. Poor API design (for example, months start with 1 and days start with 0)

2.Not thread safe
3.No consistency within the java.util and java.sql classes
4.No support for internationalization and time zones

The new Date and Time API was introduced in Java SE 8 with the following solutions:

1.The improved API design clearly separates the human-readable date time and machine time.

2.The new Date and Time API makes all the classes immutable, which is suitable for the multithreaded environment.

3.There is more clarity in the API design level. The methods are clearly defined and perform the same action in all classes.

4.The API defines separate classes for Date, Time, DateTime, Timestamp, TimeZone, and so on. The new Date and Time API works with the ISO-8601 calendar. However, you can use the API with non-ISO calendars as well.