JAVA 1.8 features:

Interview questions on java 8 features?

A,

Here are the Java 8 interview questions categorized by basic, intermediate, and advanced levels, along with their answers:

Basic Questions

1. What is the purpose of introducing Java 8?

Answer:

Java 8 was introduced to enhance the language with functional programming capabilities, improve developer productivity, and simplify complex tasks such as handling collections, date/time management, and concurrent programming. Some of its major features include lambda expressions, streams, Optional, and the new date-time API.

2. What is a lambda expression? Provide a basic syntax and example.

Answer:

A lambda expression is a shorthand for writing anonymous methods (functions) in Java. It allows you to pass behavior (code) as a parameter to a method.

Syntax:

(parameters) -> { body }

Example:

Runnable r = () -> System.out.println("Hello, Lambda!");

r.run();

3. What is a functional interface, and why is it important in Java 8?

Answer:

A functional interface is an interface with exactly one abstract method, making it eligible for lambda expressions. It allows developers to pass behavior (as lambda expressions) to methods that accept functional interfaces.

Example:

@FunctionalInterface

interface Calculator {

int add(int a, int b);

}

4. What are default methods in interfaces, and why were they introduced?

Answer:

Default methods allow an interface to provide a default implementation for a method. They were introduced in Java 8 to support backward compatibility. Now, interfaces can evolve by adding new methods without breaking existing implementations.

Example:

interface MyInterface {

default void show() {

System.out.println("Default Method");

}

}

5. What is the difference between Optional.of(), Optional.ofNullable(), and Optional.empty()?

Answer:

• Optional.of(value): Throws NullPointerException if value is null.

• Optional.ofNullable(value): Returns an empty Optional if value is null, otherwise wraps the value.

• Optional.empty(): Returns an empty Optional.

Example:

Optional<String> opt1 = Optional.of("Hello");

Optional<String> opt2 = Optional.ofNullable(null);

Optional<String> opt3 = Optional.empty();

6. What are streams in Java 8? Provide a basic example of stream operations.

Answer:

Streams are sequences of elements that support various operations (such as map, filter, reduce) to process data in a functional style. Stream operations can be intermediate (like map) or terminal (like collect).

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()

.filter(n -> n % 2 == 0)

.forEach(System.out::println);

7. How is forEach() different from a traditional for loop?

Answer:

• forEach() is a method in the Stream interface and is used to iterate over elements in a stream in a functional way. It allows you to perform actions on each element using a lambda expression.

• A traditional for loop requires explicit index management and is more imperative.

Example using forEach():

List<Integer> numbers = Arrays.asList(1, 2, 3);

numbers.forEach(System.out::println);

8. What is the use of the @FunctionalInterface annotation?

Answer:

@FunctionalInterface is used to indicate that an interface is meant to be functional, i.e., it should have exactly one abstract method. It helps ensure that the interface meets the requirement for lambda expressions. If the interface has more than one abstract method, a compile-time error will be generated.

Example:

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod();

}

9. What is a predicate in Java 8? Provide an example.

Answer:

A Predicate is a functional interface in Java 8 that represents a boolean-valued function of one argument. It is commonly used to evaluate conditions.

Example:

Predicate<Integer> isEven = n -> n % 2 == 0;

System.out.println(isEven.test(4)); // true

Intermediate Questions

1. What is the difference between map() and flatMap() in Java 8?

Answer:

• map(): Transforms each element of a stream into another object.

• flatMap(): Flattens nested structures (e.g., streams inside streams) into a single stream.

Example:

List<List<Integer>> numbers = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));

numbers.stream()

.flatMap(List::stream)

.forEach(System.out::println); // prints 1, 2, 3, 4

2. What is the purpose of Optional in Java 8, and how is it used?

Answer:

Optional is used to represent a value that may or may not be present, helping to avoid NullPointerException. It provides methods like ifPresent(), orElse(), and map() to handle missing values.

Example:

Optional<String> name = Optional.ofNullable(null);

name.ifPresent(System.out::println); // does nothing

String result = name.orElse("Default Name");

System.out.println(result); // prints "Default Name"

3. What are the advantages of using streams over traditional loops?

Answer:

• Streams provide a more declarative approach to process data.

• Operations are often easier to read and maintain.

• Streams can be parallelized for performance improvements.

• Stream operations are often chained and allow for functional-style transformations.

4. What is the difference between forEach() and collect() in Java 8 streams?

Answer:

• forEach(): A terminal operation that performs an action on each element of a stream (e.g., printing elements).

• collect(): A terminal operation that transforms a stream into a different form, such as a list or a set.

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3);

numbers.stream().collect(Collectors.toList()); // Collects elements into a list

Advanced Questions

1. What is the difference between map() and reduce() in Java 8?

Answer:

• map(): Transforms each element of a stream into another object.

• reduce(): Combines the elements of a stream into a single result by repeatedly applying a binary operation.

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

int sum = numbers.stream().reduce(0, (a, b) -> a + b); // Result: 10

2. What is method reference in Java 8?

Answer:

Method references provide a way to refer to a method of a class or object without invoking it. It simplifies lambda expressions. There are four types:

• Static method reference: ClassName::staticMethod

• Instance method reference: object::instanceMethod

• Constructor reference: ClassName::new

• Instance method reference (via class type): ClassName::instanceMethod

Example:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(System.out::println); // Using method reference instead of lambda expression

3. How do you create a parallel stream in Java 8, and what are its benefits?

Answer:

You can create a parallel stream using parallelStream() on a collection or stream().parallel() on a stream. Parallel streams allow for multi-core CPU utilization and can improve performance for large data sets, but they may not always result in performance improvement due to overhead in parallelization.

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

numbers.parallelStream().forEach(System.out::println);

4. Explain the concept of “lazy evaluation” in Java 8 streams.

Answer:

Lazy evaluation in Java 8 means that the processing of elements in a stream is deferred until a terminal operation (such as forEach(), collect()) is invoked. Intermediate operations like filter() and map() are not executed until they are necessary.

Let me know if you need further clarifications or examples!

Interview questions on functional interface?

A,

Here are some Java interview questions related to Functional Interfaces categorized into Basic, Intermediate, and Advanced levels:

Basic Level Questions

1. What is a functional interface in Java?

Answer: A functional interface is an interface that has exactly one abstract method. It can contain multiple default or static methods but must have exactly one abstract method.

2. Can you give an example of a functional interface in Java?

Answer:

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod(); // One abstract method

}

3. What is the purpose of the @FunctionalInterface annotation in Java?

Answer: The @FunctionalInterface annotation is used to indicate that an interface is intended to be a functional interface. It helps the compiler verify that the interface contains only one abstract method.

4. Can a functional interface have multiple abstract methods?

Answer: No, a functional interface can only have one abstract method. However, it can have multiple default and static methods.

Intermediate Level Questions

1. What are the main differences between a regular interface and a functional interface?

Answer:

• A regular interface can have any number of abstract methods.

• A functional interface must have exactly one abstract method but can have multiple default or static methods.

2. How do lambda expressions work with functional interfaces?

Answer: Lambda expressions provide an implementation for the abstract method in a functional interface. A lambda expression simplifies the syntax and allows you to pass behavior as arguments to methods or store it as variables.

Example:

@FunctionalInterface

interface Calculator {

int add(int a, int b);

}

public class Main {

public static void main(String[] args) {

Calculator calc = (a, b) -> a + b; // Lambda expression

System.out.println(calc.add(5, 3)); // Output: 8

}

}

3. What is the difference between a functional interface and an anonymous class?

Answer:

• A functional interface is an interface with one abstract method and is typically used with lambda expressions.

• An anonymous class is a way to implement an interface or subclass a class without giving it a name.

Example:

// Anonymous class

Runnable r1 = new Runnable() {

@Override

public void run() {

System.out.println("Hello from anonymous class");

}

};

// Functional interface (with lambda expression)

Runnable r2 = () -> System

Interview questions on lambda expressions?

A,

Here are some interview questions on Lambda expressions categorized into Basic, Intermediate, and Advanced levels:

Basic Level Questions

1. What is a Lambda expression in Java?

Answer: A lambda expression is a concise way to represent an anonymous function (a function without a name) that can be passed around and executed. It is primarily used to define the behavior of functional interfaces.

Syntax:

(parameter) -> expression

Example:

// A simple lambda expression that adds two numbers

(int a, int b) -> a + b;

2. How do you write a lambda expression for a functional interface?

Answer: A lambda expression is written to implement the single abstract method (SAM) of a functional interface.

Example:

@FunctionalInterface

interface Calculator {

int add(int a, int b); // Single abstract method

}

public class Main {

public static void main(String[] args) {

Calculator addNumbers = (a, b) -> a + b; // Lambda expression

System.out.println(addNumbers.add(5, 3)); // Output: 8

}

}

3. What are the advantages of using Lambda expressions in Java?

Answer:

• Conciseness: Lambda expressions allow you to write more compact and readable code.

• No boilerplate code: Reduces the need for creating anonymous classes.

• Improved parallel processing: Makes working with the Java Streams API easier for processing collections in parallel.

4. How do you pass a lambda expression as a parameter to a method?

Answer: A lambda expression can be passed as an argument to methods that accept functional interfaces.

Example:

interface Greet {

void greetMessage(String name);

}

public class Main {

public static void main(String[] args) {

printGreeting((name) -> System.out.println("Hello, " + name));

}

static void printGreeting(Greet greet) {

greet.greetMessage("Sandeep");

}

}

Intermediate Level Questions

1. Can you explain how lambda expressions work with the Comparator interface?

Answer: Lambda expressions can be used with the Comparator interface to define custom sorting logic in a concise manner.

Example:

import java.util.\*;

public class Main {

public static void main(String[] args) {

List<String> list = Arrays.asList("Banana", "Apple", "Orange");

// Using lambda expression to sort the list

Collections.sort(list, (str1, str2) -> str1.compareTo(str2));

System.out.println(list); // Output: [Apple, Banana, Orange]

}

}

2. What is the difference between a lambda expression and an anonymous class?

Answer:

• A lambda expression is a shorter way of writing code that implements a functional interface. It does not require a class or a method declaration.

• An anonymous class is a way to provide an implementation for an interface or abstract class without defining a named class.

Example (Anonymous Class vs Lambda):

// Anonymous Class

Runnable r1 = new Runnable() {

public void run() {

System.out.println("Runnable with anonymous class");

}

};

// Lambda Expression

Runnable r2 = () -> System.out.println("Runnable with lambda expression");

3. Can you use multiple parameters in a lambda expression?

Answer: Yes, a lambda expression can accept multiple parameters, and they are separated by commas.

Example:

// Lambda expression with two parameters

BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

System.out.println(add.apply(2, 3)); // Output: 5

4. What is the purpose of this keyword inside a lambda expression?

Answer: The this keyword inside a lambda expression refers to the instance of the enclosing class, not the lambda itself.

Example:

class MyClass {

void test() {

MyFunctionalInterface obj = () -> {

System.out.println(this.getClass().getName());

};

obj.show();

}

}

@FunctionalInterface

interface MyFunctionalInterface {

void show();

}

Advanced Level Questions

1. Can you use a lambda expression to override the toString() method?

Answer: No, you cannot override methods like toString() using lambda expressions because lambda expressions are specifically used for implementing functional interfaces (interfaces with exactly one abstract method).

2. What is the significance of the final keyword in lambda expressions?

Answer: In lambda expressions, the local variables that are used within the body of the lambda expression must be effectively final. This means that their value cannot be modified after they are assigned.

Example:

public class Main {

public static void main(String[] args) {

int num = 10; // Effectively final

Runnable r = () -> System.out.println(num); // Valid

// num = 20; // Compilation error: local variable num referenced from a lambda expression must be final or effectively final

}

}

3. Can lambda expressions be used with method references?

Answer: Yes, lambda expressions can be replaced with method references when the lambda expression is simply calling a method that matches the signature of the abstract method in the functional interface.

Example (Method Reference):

public class Main {

public static void main(String[] args) {

// Using Lambda expression

List<String> list = Arrays.asList("apple", "banana", "cherry");

list.forEach(s -> System.out.println(s.toUpperCase()));

// Using Method Reference

list.forEach(System.out::println); // Method reference

}

}

4. How does Java 8 handle scope and access to local variables inside lambda expressions?

Answer: Lambda expressions have access to effectively final local variables, fields, and parameters of the enclosing method or class. This is similar to how anonymous classes can access final variables.

5. How does the Stream API benefit from lambda expressions in Java?

Answer: The Stream API allows you to use lambda expressions for declarative operations on collections. The use of lambda expressions makes code more concise and expressive when processing collections in parallel or sequentially.

Example:

List<String> list = Arrays.asList("apple", "banana", "cherry");

list.stream()

.filter(s -> s.startsWith("a"))

.forEach(System.out::println); // Output: apple

These questions provide a comprehensive understanding of lambda expressions and their use in Java. They range from simple syntax questions to more complex scenarios involving method references, the

1.Interface(can define methods inside interface):

Let’s break this down step by step to understand the rules regarding default methods, static methods, and method resolution in Java 8.

Rule 1: Class methods take precedence over interface methods

If a class implements an interface and there is a method conflict (a method with the same name exists in both the class and the interface), the method from the class takes precedence over the default method in the interface. This is because class methods are considered more specific.

Example:

interface A {

default void show() {

System.out.println("Default show from Interface A");

}

}

class B {

public void show() {

System.out.println("Show method from Class B");

}

}

class C extends B implements A {

// No need to override show() because class B's method takes precedence

}

public class Main {

public static void main(String[] args) {

C obj = new C();

obj.show(); // Output: Show method from Class B

}

}

Rule 2: Conflict between multiple interfaces requires overriding

If a class implements multiple interfaces and both interfaces have a default method with the same name and parameters, the class must override the conflicting method. Java does not allow the ambiguity to persist.

Example:

interface A {

default void show() {

System.out.println("Default show from Interface A");

}

}

interface B {

default void show() {

System.out.println("Default show from Interface B");

}

}

class C implements A, B {

@Override

public void show() {

System.out.println("Overridden show method in Class C");

}

}

public class Main {

public static void main(String[] args) {

C obj = new C();

obj.show(); // Output: Overridden show method in Class C

}

}

Rule 3: Static methods in interfaces cannot be overridden

Static methods in interfaces belong to the interface itself and cannot be overridden or accessed through the implementing class. They are accessed directly using the interface name.

Example:

interface A {

static void show() {

System.out.println("Static show from Interface A");

}

}

class B implements A {

// Cannot override static methods

}

public class Main {

public static void main(String[] args) {

A.show(); // Output: Static show from Interface A

}

}

Rule 4: Overriding behavior when a method exists in the class and the interface

If a method exists in both the class and the interface, the following rules apply:

1. The class method overrides the interface default method.

2. If you try to explicitly override a class method in the interface using the default keyword, it will result in a compile-time error, because interfaces cannot override methods from a class.

Example:

class B {

public void show() {

System.out.println("Show method from Class B");

}

}

interface A {

default void show() {

System.out.println("Default show from Interface A");

}

}

class C extends B implements A {

// No need to override show() because Class B already provides it

}

public class Main {

public static void main(String[] args) {

C obj = new C();

obj.show(); // Output: Show method from Class B

}

}

Rule 5: Cannot override parameters in interfaces

You cannot declare a method in an interface that “overrides” a class method but with different parameters (method overloading). If a class method exists, the interface can only define methods with the same signature to avoid conflict. Trying to define a method with a different signature in the interface will result in ambiguity or a compile-time error.

Example:

class B {

public void show(String msg) {

System.out.println("Show method from Class B: " + msg);

}

}

interface A {

// This would cause a conflict because it doesn't match the method in Class B

// default void show() {

// System.out.println("Default show from Interface A");

// }

}

public class Main {

public static void main(String[] args) {

// Compile-time error if such a conflict exists

}

}

Summary of Key Rules:

1. Class methods take precedence over interface methods, even if the interface has a default method.

2. If multiple interfaces have conflicting default methods, the implementing class must override the method to resolve ambiguity.

3. Static methods in interfaces cannot be overridden and must be accessed using the interface name.

4. Interfaces cannot override class methods with different parameters, as it violates Java’s overriding rules.

Would you like further clarification or examples?

Abstact method:

If you define method in interface what is deffrence for abstract and this.(for that only we use default and static to separate)

What is abstract class.(in this methods can be declared with or without body in it.)

How multiple inheritance achieved in this.(ex:interface a implements b,c now if you call show which exists in both classes it gives error to overcome that declare it in the interface a itself so error will be rectified).

package com.lambda;

interface Phone {

void call();

default void message() {

System.out.println("sent");

}

}

class AndroidPhone implements Phone {

public void call() {

System.out.println("Calling");

}

}

public class DemoInterface {

public static void main(String[] args) {

Phone p = new AndroidPhone();

p.call();

p.message();

}

}

Static methods:call interface methods without creating object for interface.

package com.lambda;

interface Phone {

void call();

static void message() {

System.out.println("sent");

}

}

class AndroidPhone implements Phone {

public void call() {

System.out.println("Calling");

}

}

public class DemoInterface {

public static void main(String[] args) {

//Phone p = new AndroidPhone();//no need to create this method because static can call directlt.

Phone.message();//its called without creating object.

}

}

Default methods can have a body and be overridden by implementing classes.

• Static methods belong to the interface and are not inherited by implementing classes.

2.FOR each method:

import java.util.Arrays;

import java.util.List;

// foreach Method in Java 8

// External Loops

// Internal Loops - Java 8

public class DemoForEach {

public static void main(String args[]) {

List<Integer> values = Arrays.asList(4, 5, 6, 7, 8);

// External loop using a traditional for loop

for (int i = 0; i < values.size(); i++) {

System.out.println(values.get(i));

}

// Enhanced for loop

for (int i : values) {

System.out.println(i);

}

// Internal loop using forEach with a lambda expression

values.forEach(i -> System.out.println(i));

}

}

3.LAMBDA expression.(overriding interface method using lambdexpresaion while creating object).

How to use lambda expression:

A lambda expression in Java is a concise way to represent an anonymous function (a block of code) that can be passed as an argument to methods or stored as variables. Introduced in Java 8, lambda expressions make it easier to implement functional programming concepts in Java.

Syntax of Lambda Expressions

(parameters) -> expression\_or\_block

• Parameters: List of input parameters for the function. These are optional if no arguments are required.

• Arrow (->): Separates the parameters from the body of the lambda expression.

• Body: The code block or expression that gets executed.

Examples

1. Without Parameters

Runnable r = () -> System.out.println("Hello, World!");

r.run(); // Output: Hello, World!

2. With One Parameter (No Parentheses Required)

interface Greeting {

void say(String message);

}

Greeting g = message -> System.out.println("Message: " + message);

g.say("Hi!"); // Output: Message: Hi!

3. With Multiple Parameters

interface MathOperation {

int operate(int a, int b);

}

MathOperation add = (a, b) -> a + b;

System.out.println(add.operate(5, 3)); // Output: 8

4. With a Code Block

MathOperation subtract = (a, b) -> {

System.out.println("Subtracting...");

return a - b;

};

System.out.println(subtract.operate(5, 3)); // Output: Subtracting... 2

Functional Interfaces and Lambda Expressions

A lambda expression can only be used with functional interfaces—interfaces with a single abstract method. Common examples of functional interfaces in Java:

1. Runnable: void run()

2. Callable: V call()

3. Comparator: int compare(T o1, T o2)

4. Custom Functional Interfaces: You can define your own functional interface using the @FunctionalInterface annotation.

Example:

@FunctionalInterface

interface Printer {

void print(String message);

}

Printer printer = message -> System.out.println(message);

printer.print("Hello Lambda!"); // Output: Hello Lambda!

Benefits of Lambda Expressions

1. Concise Code: Avoids verbosity compared to anonymous classes.

2. Improved Readability: Makes the code more readable, especially for short implementations.

3. Supports Functional Programming: Enables a more functional programming style in Java.

4. Stream API Integration: Simplifies working with collections using methods like forEach, map, and filter.

Using Lambda with Stream API

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Use a lambda expression with forEach

names.forEach(name -> System.out.println(name));

// Use a lambda expression with filter

names.stream()

.filter(name -> name.startsWith("A"))

.forEach(System.out::println); // Output: Alice

}

}

Limitations of Lambda Expressions

1. Can Only Be Used with Functional Interfaces: They rely on interfaces with exactly one abstract method.

2. Not Suitable for Complex Logic: For large blocks of logic, named classes or methods are more readable.

3. Difficult Debugging: Errors in lambda expressions can sometimes be harder to debug.

Would you like examples or a deeper explanation of specific use cases?

Your code demonstrates three approaches to implementing and using an interface (a) with a single method (show) in Java:

1. Using a concrete class (b) that implements the interface.

2. Using an anonymous inner class.

3. Using a lambda expression (since a is a functional interface).

Below is a detailed explanation of each approach, along with a cleaned-up version of your code for better readability:

Code

package org.example;

import java.util.Arrays;

import java.util.List;

public class Main {

// Functional interface

interface a {

void show(); // Abstract method

}

// Concrete class implementing the interface

public static class b implements a {

@Override

public void show() {

System.out.println("show");

}

}

public static void main(String[] args) {

// 1. Using a concrete class

a obj;

obj = new b(); // Class 'b' provides implementation

obj.show(); // Output: show

// 2. Using an anonymous inner class

a obj1 = new a() {

@Override

public void show() {

System.out.println("HI");

}

};

obj1.show(); // Output: HI

// 3. Using a lambda expression

a obj3 = () -> System.out.println("hi sandeep");

obj3.show(); // Output: hi sandeep

// Additional demonstration of lambda in collections

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5);

l.forEach(i -> System.out.println(i)); // Prints each element

}

}

Explanation of Approaches

1. Concrete Class Implementation

• The class b implements the interface a and provides a concrete definition for the show() method.

• The object obj is created using the concrete class b, and when obj.show() is called, it uses the show() implementation from b.

Output:

show

2. Anonymous Inner Class

• An anonymous inner class is used to provide an inline implementation of the interface a.

• Here, obj1 is an instance of an anonymous subclass of a that overrides the show() method.

Output:

HI

3. Lambda Expression

• Since the interface a is a functional interface (i.e., it has a single abstract method), you can use a lambda expression to define the show() method’s behavior directly.

• This approach is concise and avoids creating a separate class or an anonymous inner class.

Output:

hi sandeep

Additional: Using Lambdas with Collections

• The forEach method of the List interface accepts a lambda expression to process each element of the list.

• In this case, each element of the list l is printed using a lambda expression.

Output (List Elements):

1

2

3

4

5

Key Points

1. Functional Interfaces:

• An interface with a single abstract method (like a) is called a functional interface.

• Functional interfaces are the foundation for lambda expressions in Java.

2. Lambda Expressions:

• Syntax: (parameters) -> {body}.

• Lambdas are used to directly provide behavior without defining a concrete class or using an anonymous inner class.

3. Anonymous Inner Class vs Lambda:

• Anonymous inner classes can implement multiple methods (if needed for non-functional interfaces), but they are verbose.

• Lambdas are concise and preferred for functional interfaces.

4. List Operations with Lambdas:

• Functional-style programming in Java allows processing collections using methods like forEach, map, and filter in combination with lambdas.

Sample Output

show

HI

hi sandeep

1

2

3

4

5

What are functional interfaces?

A,

A functional interface in Java is an interface that contains exactly one abstract method. It can have other methods like default or static methods, but it must have only one abstract method.

Functional interfaces are the foundation for lambda expressions in Java, as they allow you to define a function in a concise way without explicitly implementing the interface.

Key Characteristics

1. Single Abstract Method: A functional interface has exactly one abstract method. This is why it is also known as a SAM (Single Abstract Method) Interface.

2. Default and Static Methods: Functional interfaces can have default or static methods, but they do not affect the “functional” aspect of the interface.

3. @FunctionalInterface Annotation: Although optional, it’s recommended to use the @FunctionalInterface annotation. It ensures that the interface adheres to the functional interface rules and prevents accidental addition of extra abstract methods.

Examples of Functional Interfaces in Java

1. Predefined Functional Interfaces

Java provides many built-in functional interfaces in the java.util.function package, such as:

1. Runnable:

Abstract Method: void run().

Example:

Runnable r = () -> System.out.println("Running");

r.run(); // Output: Running

2. Supplier:

Abstract Method: T get().

Example:

Supplier<String> supplier = () -> "Hello, Functional Interface!";

System.out.println(supplier.get()); // Output: Hello, Functional Interface!

3. Consumer:

Abstract Method: void accept(T t).

Example:

Consumer<String> consumer = message -> System.out.println("Message: " + message);

consumer.accept("Hi there!"); // Output: Message: Hi there!

4. Function<T, R>:

Abstract Method: R apply(T t).

Example:

Function<Integer, Integer> square = x -> x \* x;

System.out.println(square.apply(5)); // Output: 25

5. Predicate:

Abstract Method: boolean test(T t).

Example:

Predicate<Integer> isEven = x -> x % 2 == 0;

System.out.println(isEven.test(4)); // Output: true

System.out.println(isEven.test(5)); // Output: false

2. Custom Functional Interface

You can define your own functional interface.

@FunctionalInterface

interface Greeting {

void sayHello(String name); // Single Abstract Method

}

public class Main {

public static void main(String[] args) {

Greeting greeting = name -> System.out.println("Hello, " + name + "!");

greeting.sayHello("Sandeep"); // Output: Hello, Sandeep!

}

}

Default and Static Methods in Functional Interfaces

Functional interfaces can include default and static methods without violating the “functional” aspect.

@FunctionalInterface

interface MathOperation {

int operate(int a, int b); // Single Abstract Method

// Default method

default void printMessage() {

System.out.println("Performing operation...");

}

// Static method

static void description() {

System.out.println("This is a functional interface for mathematical operations.");

}

}

public class Main {

public static void main(String[] args) {

MathOperation addition = (a, b) -> a + b;

MathOperation.description(); // Output: This is a functional interface for mathematical operations.

addition.printMessage(); // Output: Performing operation...

System.out.println(addition.operate(5, 3)); // Output: 8

}

}

@FunctionalInterface Annotation

The @FunctionalInterface annotation is used to indicate that an interface is intended to be a functional interface. If you try to add more than one abstract method, the compiler will throw an error.

Example:

@FunctionalInterface

interface MyFunctionalInterface {

void execute(); // Single Abstract Method

// Uncommenting this will cause a compile-time error:

// void anotherMethod();

}

Why Use Functional Interfaces?

1. Enable Lambda Expressions: Functional interfaces are required to use lambda expressions.

2. Simplify Code: They reduce the need for verbose anonymous inner classes.

3. Built-In Functional Interfaces: Java provides many predefined functional interfaces for common use cases (e.g., Runnable, Supplier, Consumer, etc.).

4. Support Functional Programming: They allow Java to adopt functional programming principles.

4.Consumer interface.

Code below:

While implementing interface consumer than make sure in that class you pass meyhod with

\*\*Show(integer i) not show(int i);

Code 1:

class sandeep implements Consumer<Integer> {  
 @Override  
 public void accept(Integer i) {  
 System.*out*.println(i);  
 }  
}  
public class Consumerinterface {  
  
this is optional  
/\*public static class sandeep implements Consumer<Integer> {  
 @Override  
 public void accept(Integer i) {  
 System.*out*.println(i);  
 }  
}\*/

public static void main(String[] args){  
 System.*out*.println("hi");  
 List<Integer> l= Arrays.*asList*(1,2,3,4,5);  
 Consumer<Integer> c=new sandeep();  
 l.forEach(c);  
  
 }  
}

Code 2:

package org.example;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Consumer;  
  
  
  
public class Consumerinterface {  
  
  
  
  
  
 public static void main(String[] args){  
 System.*out*.println("hi");  
 List<Integer> l= Arrays.*asList*(1,2,3,4,5);  
 /\* Consumer<Integer> c=new Consumer<Integer>() {  
 @Override  
 public void accept(Integer i) {  
 System.out.println(i);  
 }  
 };\*/  
 Consumer<Integer> c=(Integer i) ->{  
 System.*out*.println(i);  
 };  
   
 l.forEach(c);  
  
 }  
}

6.Time Api:

Code:

package org.example;  
  
import java.time.LocalDate;  
import java.time.LocalTime;  
import java.time.Month;  
  
public class Date {  
 public static void main(String[] args) {  
 LocalTime d= LocalTime.*now*();  
 LocalDate d1= LocalDate.*now*();  
 LocalDate d2=LocalDate.*of*(1999, Month.*FEBRUARY*,28);  
 System.*out*.println(d1);  
 System.*out*.println(d);  
 System.*out*.println(d2);  
  
  
 }  
}

interview questions by backend developer?

Here are the answers to the questions you provided regarding Java 8:

**Q1. What are new features of Java 8?**

Java 8 introduced several major features, including:

1. **Lambda Expressions**: To provide clear and concise syntax for writing code in functional programming style.
2. **Streams API**: To perform operations on data in a functional style.
3. **Default Methods in Interfaces**: To allow methods in interfaces to have a default implementation.
4. **Functional Interfaces**: Interfaces with a single abstract method, enabling the use of lambda expressions.
5. **Method References**: A shorthand notation for calling a method.
6. **Optional Class**: To avoid NullPointerException.
7. **New Date and Time API**: To handle date and time more effectively (e.g., java.time package).
8. **Nashorn JavaScript Engine**: To allow integration with JavaScript code.
9. **New Stream Operations**: Such as map, flatMap, filter, collect, etc.
10. **Concurrent API Enhancements**: Enhanced APIs for concurrency, including CompletableFuture.

**Q2. What is the default method in Java 8?**

A **default method** in Java 8 is a method in an interface that has a default implementation. It allows an interface to have methods with a body (implementation), so existing interfaces can evolve without breaking existing implementations.

interface MyInterface {

default void defaultMethod() {

System.out.println("Default Method");

}

}

**Q3. What is method reference in Java 8?**

**Method references** provide a shorthand way of calling methods via lambda expressions. Instead of writing a lambda expression that calls a method, you can use a method reference.

Syntax:

* ClassName::methodName
* instance::methodName

Example:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(System.out::println); // Method reference

**Q4. What are Types of method reference in Java 8?**

There are four types of method references in Java 8:

1. **Reference to a static method**:  
   ClassName::staticMethodName
2. **Reference to an instance method of a particular object**:  
   object::instanceMethodName
3. **Reference to an instance method of an arbitrary object of a particular type**:  
   ClassName::instanceMethodName
4. **Reference to a constructor**:  
   ClassName::new

**Q5. What is Metaspace in Java 8?**

In Java 8, **Metaspace** replaces the **PermGen** (Permanent Generation) space in the JVM. It is used to store class metadata (e.g., class definitions, methods, etc.). The major difference is that Metaspace is allocated from native memory rather than the heap, which allows for more dynamic and efficient memory usage. The size of Metaspace is limited only by the system's physical memory.

**Q6. What is an Optional class in Java 8?**

The **Optional** class in Java 8 is used to represent a value that may or may not be present. It helps to prevent NullPointerExceptions by providing methods to handle null values gracefully.

Example:

Optional<String> name = Optional.ofNullable("Alice");

name.ifPresent(System.out::println); // prints "Alice"

**Q7. How do we avoid NullPointerException in Java 8?**

To avoid NullPointerException in Java 8:

1. **Use Optional**: Wrap values in Optional to handle nulls.
2. **Use method references**: Optional methods like ifPresent, orElse, orElseGet, map, etc., avoid the need to check for null.

Example:

Optional<String> name = Optional.ofNullable("Alice");

String result = name.orElse("Default Name"); // Avoids null handling explicitly

**Q8. What are the different methods in the Optional class that you have used?**

Common methods in the Optional class:

1. of(T value) – Returns an Optional with a non-null value.
2. ofNullable(T value) – Returns an Optional, which can be null.
3. ifPresent(Consumer<? super T> action) – Executes the provided action if the value is present.
4. orElse(T other) – Returns the value if present, otherwise returns the default value.
5. map(Function<? super T,? extends U> mapper) – Transforms the value if present.
6. flatMap(Function<? super T,Optional<U>> mapper) – Similar to map, but allows for returning an Optional itself.
7. filter(Predicate<? super T> predicate) – Filters the value based on the condition.

**Q9. What are Functional Interfaces in Java 8?**

A **functional interface** is an interface with only one abstract method, but it can have multiple default or static methods. They are the foundation of lambda expressions in Java.

Example:

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod();

}

**Q10. Difference between functional interface and abstract class that you have used?**

* **Functional Interface**:
  1. Can have only one abstract method.
  2. Can have multiple default/static methods.
  3. Designed for use with lambda expressions.
* **Abstract Class**:
  1. Can have multiple abstract methods.
  2. Can have fields and constructors.
  3. Can provide a partial implementation.

Functional interfaces focus on behavior, while abstract classes can represent both behavior and state.

**Q11. What is Lambda Expression in Java 8?**

A **lambda expression** is a concise way to express instances of single-method interfaces (functional interfaces). It provides a clear and succinct syntax for writing anonymous methods (functions).

Syntax:

(parameter) -> expression

Example:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name)); // Lambda expression

**Q12. When to use map and flatmap?**

* **map**: Used when you want to transform each element in a stream into another value.
* **flatMap**: Used when you want to transform each element into a stream of values and then flatten all those streams into one.

Example:

// map

List<String> names = Arrays.asList("Alice", "Bob");

List<Integer> lengths = names.stream().map(String::length).collect(Collectors.toList());

// flatMap

List<List<Integer>> lists = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));

List<Integer> flatList = lists.stream().flatMap(List::stream).collect(Collectors.toList());

**Q13. What is new in date and time changes in Java 8?**

Java 8 introduced a new **Date and Time API** (java.time) to handle date and time more effectively. This new API is immutable, thread-safe, and much easier to work with compared to the old java.util.Date and Calendar classes.

Key classes:

1. LocalDate – Represents a date without a time zone.
2. LocalTime – Represents a time without a date.
3. LocalDateTime – Represents both date and time.
4. ZonedDateTime – Represents a date and time with a time zone.
5. Instant – Represents a specific moment in time.

**Q14. Why we need lambda expression?**

Lambda expressions enable **functional programming** in Java, making it possible to pass behavior (functions) as arguments. They help reduce boilerplate code (like anonymous inner classes) and enable the use of functional-style operations like map, filter, reduce, etc., on collections or streams.

**Q15. How lambda expression and functional interfaces are related?**

Lambda expressions can only be used with **functional interfaces** (interfaces with one abstract method). Lambda expressions provide a way to implement the abstract method of a functional interface concisely.

**Q16. Will lambda expression create an object whenever it’s executed?**

Lambda expressions themselves do not necessarily create a new object each time they are executed. A **lambda expression** is typically implemented as an instance of a **functional interface**, but the JVM may reuse the same object for efficiency.

**Q17. Which are the existing Functional Interfaces before Java 8?**

Before Java 8, there were some existing functional interfaces like Runnable, Callable, and Comparator. However, Java 8 introduced many more built-in functional interfaces in the java.util.function package, such as Predicate, Function, Consumer, Supplier, etc.

**Q18. How can you optimize a stream pipeline to improve memory efficiency when working with large datasets?**

To optimize stream pipelines for memory efficiency:

1. **Use Lazy Evaluation**: Streams are lazily evaluated, meaning operations are only performed when required. Avoid eager operations like collect until necessary.
2. **Use Primitive Streams**: Use IntStream, LongStream, and DoubleStream to reduce the overhead of boxing/unboxing.
3. **Parallel Streams**: In certain scenarios, parallel streams can improve performance by using multiple threads, but ensure it's safe to parallelize.

**Q19. Explain how BiFunction, BiConsumer, and BiPredicate are different from their single-parameter counterparts, and give a scenario where you would use them?**

* **BiFunction**: Takes two arguments and produces a result. Used when you need to combine two values into a new result. Example: BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
* **BiConsumer**: Takes two arguments and returns void. Used when you need to perform an action on two values. Example: BiConsumer<Integer, Integer> printSum = (a, b) -> System.out.println(a + b);
* **BiPredicate**: Takes two arguments and returns a boolean. Used for filtering or testing conditions on two values. Example: BiPredicate<Integer, Integer> isEqual = (a, b) -> a.equals(b);

**Q20. How would you implement a custom FunctionalInterface that takes three arguments and returns a result? How does this differ from built-in interfaces?**

You can define a custom functional interface as follows:

@FunctionalInterface

interface TriFunction<T, U, V, R> {

R apply(T t, U u, V v);

}

This differs from built-in interfaces because Java 8’s java.util.function package does not have an interface for three arguments. You can create your own custom interfaces to suit specific needs.

**Q21. In what scenarios might you prefer using method references (::) over lambda expressions in Java 8? Can you provide a performance comparison between the two?**

* **Use Method References**: When the lambda expression simply calls an existing method, method references make the code more readable and concise.
* **Performance**: There’s no significant performance difference between method references and lambda expressions as both compile to the same bytecode. The choice depends more on readability and coding style.

**Q22. Explain the use of the BinaryOperator interface. When would you choose BinaryOperator over BiFunction?**

BinaryOperator<T> is a specialization of BiFunction<T, T, T> where both input parameters and the result are of the same type. It’s used when you are performing an operation on two values of the same type and returning a result of the same type.

Example:

BinaryOperator<Integer> add = (a, b) -> a + b;

Use BinaryOperator when both arguments and the return type are of the same type. Use BiFunction when they are of different types.