Java 8 Interview Questions

Shristi Technology Labs

Features of Java 8

- Default and static methods
- Functional interfaces
- Lambda Expressions
- Date Time API
- Streams API

1. What are functional interfaces in Java 8?

- A functional interface has exactly one abstract method (SAM – Single Abstract Method).
  - It can have multiple default and static methods.
  - May or may not be annotated with **@FunctionalInterface**
  - Can be implemented using Lambda Expressions
  - **eg., Runnable, Callable, Comparator, Cons**

```
@FunctionalInterface
public interface ICalculator {

    double calculate(int x, int y);
}
```

```
@FunctionalInterface
public interface BonusCalculator {
    public void  calcBonus(int x);
    public default void greetMessage(){
        System.out.println(“default method ");
     }
    public static void Check(){
         System.out.println("static method ");
      }
}
```

2. What are Lambda Expressions in Java?

- is an anonymous function that helps implement functional programming.
- Is defined as a method without name, access specifier, return type.
- Make the code writing simple, short and readable.
- Can be used only with functional interfaces.

Syntax:

```
(parameters) -> {expression}
```

```
public int add(int x, int y){
  return x+y;
}
```

```
(int  x, int y) -> {
    return x+y;
 }
```

```
( x,  y) ->  x+y;
```

```
//no argument - single statement, no curly braces
  () -> System.out.println ( “Hello World”);

//one argument - single statement with curly braces
  (message ) -> { System.out.println (message) ; }

//with two argument and types
  (int a,int b ) -> { return a*b; }

//two arguments - multiple statements
  (a,b) -> {
     int sum  = a+b;
     return sum;
  }
```

3. What happens if you modify a local variable inside a Lambda?(Tricky)

- Lambda expressions can access local variables, but they must be effectively final (i.e., not modified after assignment).

- If you try to modify it, compilation will fail.

```
String message = "Hello";
IGreeter greeter = (String username) -> {
    message = "Welcome"; //local variables are by default final can't modify
    System.out.println("Great Day " + username);
};
```

4. What is the use of default methods in interfaces?

- helps to add new functionality to the existing interfaces in an application
- Adding a default method to an existing interface does not break the contract
- Default methods are implicitly public
- Helps to add a common behavior across all implementing classes of the interface

```
public interface BonusCalculator {

    void calculate(int amount);
    default void policyType(){
          System.out.println("Policy for bonus calculation");
    }
    default void greet() {
        System.out.println("Welcome");
    }

}
```

5. What will happen if two interfaces have same default method and a class implements it?(Tricky)

If two interfaces have same default method and a class implements it then => **compilation fails**

Solution: ***the implementation class must override them***

```
public interface BonusCalculator {

    void calculate(int amount);
    default void policyType(){
          System.out.println("Policy for bonus");
    }
}
```

```
@FunctionalInterface
public interface AllowanceCalculator {
    void calculate(int amount);
    default void policyType(){
          System.out.println("Policy for allowance");
    }
}
```

```
public class EmployeeDetails implements BonusCalculator,AllowanceCalculator{
    @Override
    public void calculate(int amount) {}
    @Override
    public void policyType() {
        System.out.println("policy for employees...");
        AllowanceCalculator.super.policyType();
        BonusCalculator.super.policyType();
    }
}
```

6. What is the use of static methods in interfaces?

- Can be used to provide a common functionality for all the implemetation classes
- Static methods can be called using the interface name only

```
public interface BonusCalculator {

    void calculate(int amount);
    default void policyType(){
          System.out.println("Policy for bonus calculation");
    }
    static void call() {
        System.out.println("can be called only using interface");
    }

}
```

6. What is the use of static methods in interfaces?

- Can be used to provide a common functionality for all the implemetation classes
- Static methods can be called using the interface name only

```
public interface BonusCalculator {

    void calculate(int amount);
    default void policyType(){
          System.out.println("Policy for bonus calculation");
    }
    static void call() {
        System.out.println("can be called only using interface");
    }

}
```

7. What is the difference between static and default methods in interfaces?(Tricky)

static methods

- Can be used to provide a common functionality for all the implemetation classes
- can be called using the interface name only
- cannot be overridden

default methods

- Can be used to add a new functionality to an existing interface
- can be called by the object of the implementation class
- can be overridden
- for backward compatibility

8.What are Streams in Java 8?

- A Stream represents a sequence of elements and supports parallel and aggregate operations.
- Streams are abstraction for processing collections of values.
- Streams can be created from collections, arrays, or iterators.
- A stream does not store its elements
- Stream operations don’t change their source
- Have Specialized Streams for primitive data types

9. How to create streams from an Array?

- Use **`Stream.of()`** method or **`Arrays.stream()`** method

```
String[] names = new String[] {"Ram","John","Sri"};
// create a stream from an array
Stream.of(names).forEach(name-> System.out.println(name));

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

10. How to create streams from a List?

- Use **`stream()`** method

```
List<String> courses = Arrays.asList("Java","Angular","Node");
// convert to a stream
courses.stream().forEach(System.out::println);
```

11. Difference between intermediate & terminal operations in Stream

Intermediate Operations

- Returns a new Stream
- methods are **map(), filter(), sorted(),limit**
- Are lazy operations - will be executed only when a terminal operation is invoked.

Terminal Operations

- Consumes the Stream and returns the result
- methods are **forEach(), collect(), count())**
- Are early operations

12.What is the difference between map() and flatMap() in Streams?

map() -transforms elements of one type into same/another type individually.

```
Arrays.stream(courses).map(str->str.toLowerCase())
                      .forEach(System.out::println);
System.out.println();
```

flatMap()

- flattens multiple streams into a single stream.
- is useful when dealing with nested collections,

```
// List of List of employees
List<List<Employee>> employees = Arrays.asList(
                        Arrays.asList(employee1,employee2),
                        Arrays.asList(employee3,employee4 ));

// returns a stream of list of employees as blocks
Stream<List<Employee>> empListStream = employees.stream();
// returns a stream of employees
Stream<Employee> employeeStream =  empListStream
        .flatMap(employeeList->employeeList.stream());
employeeStream.forEach(System.out::println);
```

13. What happens if you try to use a Stream after a terminal operation?(Tricky)

- It throws `java.lang.IllegalStateException: stream has already been operated upon or closed`

14. What is the difference between findFirst() and findAny() in Streams?

`findFirst()`

- returns the first element in sequential order.
- guarantees order but might be slower in parallel execution.

`findAny()`

- returns any element, especially useful for parallel streams
- is optimized for performance in parallel streams and does not guarantee order..

15. How can you create an infinite Stream in Java 8?(Tricky)

- Using **generate()** method

```
// create an infinite stream
Stream.generate(()->"Hello")
      .forEach(num->System.out.println(num));
```

16. How to remove duplicates from a List using Java 8?(Tricky)

- Using **distinct()** method

```
List<Integer> numbers = Arrays.asList(10, 12, 27, 33, 54, 44, 54,33);
List<Integer> uniqueNumbers = numbers.stream()
                                     .distinct()
                                     .collect(Collectors.toList());
System.out.println(uniqueNumbers);
```

17. How do you sort the elements in a List using Java 8?

- Using **`sorted()`** method

```
Arrays.asList("Java","Angular","CSS","Html")
.stream().sorted() .forEach(System.out::println);
```

18. How to group elements in a collections?

- **`Collectors.groupingBy()`** is used to group data.

```
//grouping by author
Map<String, List<Book>> booksByAuthor =
        books.stream()
        .collect(Collectors.groupingBy(Book::getAuthor));
System.out.println(booksByAuthor);
```

19. What is the difference between limit() and skip()?

limit(): Returns first n elements from a stream.

skip(): Skips first n elements and returns the rest.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().limit(3).forEach(System.out::print); // Output: 123
numbers.stream().skip(3).forEach(System.out::print); // Output: 45
```

20. What is Optional in Java 8 and why is it useful?

- Optional is a container object that may or may not contain a value.
- Helps to avoid NullPointerException.
- Always check the optional before printing the value
- Few methods of Optional

get(),isEmpty(), isPresent(), orElse(), orElseGet(), orElseThrow()

```
Optional<String> name = Optional.ofNullable("Great day");
System.out.println(name.orElse("Welcome"));
```

21. When to use parallel streams?(Tricky)

- Use for large datasets that require parallel execution.
- Avoid for small datasets or cases with high overhead of thread management.

```
int sum  = IntStream.range(100, 2000).parallel().sum();
System.out.println(sum);
```

21. What are primitive streams?

- Special streams for working with the primitive data types int, long and double.
  - **IntStream**
  - **LongStream**
  - **DoubleStream**
- Uses specialized lambda expressions –
  - **e.g. IntFunction , IntPredicate**
- Supports the terminal aggregate operations sum() and average()

```
IntStream stream = Arrays.stream(new int[] { 40, 20, 30, 91, 16, 76 });
int sum = stream.filter(x -> x > 20).sum();
System.out.println(sum);
```

22. Name the functional interfaces in Java 8 & their use

`Consumer<T>`

- Accepts an input but returns nothing.

`Supplier<T>`

- Returns values without taking input.

`Predicate<T>`

- Returns a boolean value.

`Function<T,R>`

- Takes an input of one type and returns an output of another/same type

23. How does forEachOrdered() behave differently from forEach() in parallel streams?(Tricky)

- `forEach()` in parallel streams does not guarantee order.
- `forEachOrdered()` maintains the original order, even in parallel streams.

```
//using forEach
IntStream.range(1, 10).parallel().forEach(n -> System.out.print(n + " "));
System.out.println();
//using forEachOrdered
IntStream.range(1, 10).parallel().forEachOrdered(n -> System.out.print(n + " "));
```

output:

```
7 3 5 6 8 2 4 9 1
1 2 3 4 5 6 7 8 9
```

24. What happens when you modify a list while iterating over it using Streams?(Tricky)

- Streams do not allow modification of the source while iterating.
- If you try to add or remove elements from the list during a stream operation, it throws `ConcurrentModificationException`.

25. What is the difference between orElse() and orElseGet()?(Tricky)

orElse() - always evaluates the fallback value, even if it's not needed.

orElseGet() - only evaluates when the value is missing, avoiding unnecessary computation.

```
Using orElse
Processing
Great Day
Using orElseGet
Great Day
```

```java
public static void main(String[] args) {
    Optional<String> opt = Optional.of("Great Day");
    System.out.println("Using orElse");
    String value1 = opt.orElse(printDefault());
    System.out.println(value1);

    System.out.println("Using orElseGet");
    String value2 = opt.orElseGet(() -> printDefault());
    System.out.println(value2);
}

static String printDefault() {
    System.out.println("Processing");
    return "welcome";
}
```

Bonus Question

What is the difference between Collectors.toMap() and Collectors.groupingBy()?

Collectors.toMap()

- converts a stream into a single map
- but it throws an exception if there are duplicate keys.

Collectors.groupingBy()

- groups elements by a key and allows multiple values per key.

What is reduce() and how does it work in Streams?

- reduce() combines elements of a stream into a single value.
- It takes an identity, an accumulator function, and a combiner (for parallel execution).