**JAVA 8**

Java 8 introduced several powerful features that transformed the way Java developers write and manage code. Some of the key features include Lambda Expressions, Streams API, Optional, default methods in interfaces, and new Date and Time API. Below are some of the most common Java 8 interview questions, including conceptual explanations and sample code snippets:

1. What are the major features introduced in Java 8?

* Lambda Expressions: Allows writing anonymous methods in a more concise and functional style.
* Streams API: Provides a new abstraction to process sequences of elements (e.g., collections) in a functional way.
* Functional Interfaces: Interfaces with just one abstract method, used primarily in lambda expressions.
* Default Methods: Methods in interfaces that have a body, allowing backward compatibility with existing interfaces.
* Optional: A container that may or may not contain a non-null value, used to avoid null pointer exceptions.
* New Date and Time API: A comprehensive API to handle date, time, and time zones, replacing the old java.util.Date and java.util.Calendar classes.
* Method References: Shorthand notation of a lambda expression that calls a method directly.

2. What is a Lambda Expression in Java 8?

Answer: A lambda expression is a concise way to represent an anonymous method (or function) using an expression rather than a block of code. Lambda expressions enable functional programming in Java.

Syntax:

java

Copy code

(parameters) -> expression

Example:

java

Copy code

// Traditional way

Runnable r = new Runnable() {

public void run() {

System.out.println("Hello, World!");

}

};

// Lambda way

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

Key Points:

* () -> {} denotes a method with no parameters that returns nothing.
* Lambdas can be used to implement functional interfaces.

3. What is a Functional Interface?

Answer: A functional interface is an interface that has only one abstract method (but can have multiple default or static methods). It can be used as the target type for lambda expressions or method references.

**Predicate<T>**

* Represents a **boolean-valued function** of one argument.
* It is used for testing conditions on a single input value and returns a boolean result.

A computer screen shot of a computer screen

Description automatically generated

**Function<T, R>**

* Represents a function that takes an input of type T and returns a result of type R.
* This is used when you want to **transform** or **map** an input to an output.

A computer screen with white text

Description automatically generated

**Consumer<T>**

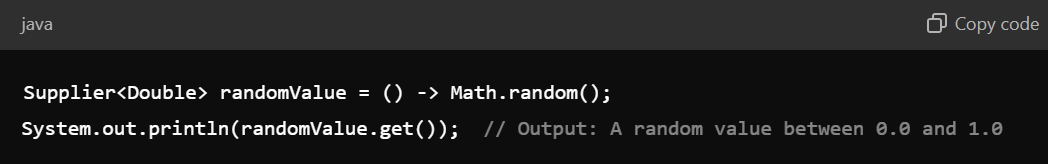
* Represents an operation that takes a single input argument of type T and returns **no result**. It is used for operations that **consume** input but don't return a result.

A screen shot of a computer

Description automatically generated

**Supplier<T>**

* Represents a function that **supplies** a result of type T without taking any input.
* It’s used when you need to generate or supply values (e.g., factory methods



**BinaryOperator<T>**

* Represents an operation on two operands of the same type that returns a result of the same type.
* It is a special case of BiFunction<T, T, T>, where the two arguments and the return type are the same.

A black and white screen with white text

Description automatically generated

**BiFunction<T, U, R>**

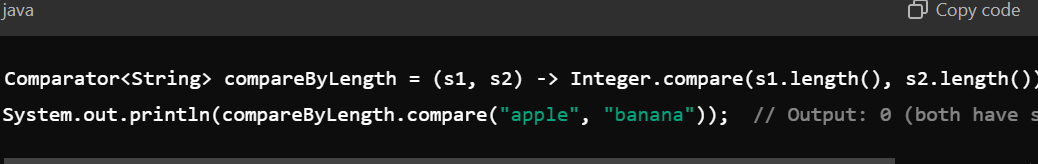
* Represents a function that takes two arguments of types T and U and returns a result of type R.

A computer screen shot of a computer code

Description automatically generated

**Comparator<T>**

* Represents a comparison function, which imposes a total ordering on a collection of objects of type T.



4. What is the Stream API in Java 8?

Answer: The Stream API provides a high-level abstraction for working with sequences of elements (e.g., collections) in a functional way. It supports operations such as filtering, mapping, and reducing.

Key Operations:

1. **Intermediate Operations**: These return a new stream and are **lazy** (i.e., they are not executed until a terminal operation is invoked).
   * Examples: filter(), map(), flatMap(), distinct(), sorted(), etc.
2. **Terminal Operations**: These trigger the processing of the stream and produce a result or a side-effect. Once a terminal operation is invoked, the stream is consumed and cannot be reused.
   * Examples: collect(), forEach(), reduce(), count(), min(), max(), etc.
     1. Filter Operation

A screen shot of a computer program

Description automatically generated

* + 1. map() - Intermediate Operation



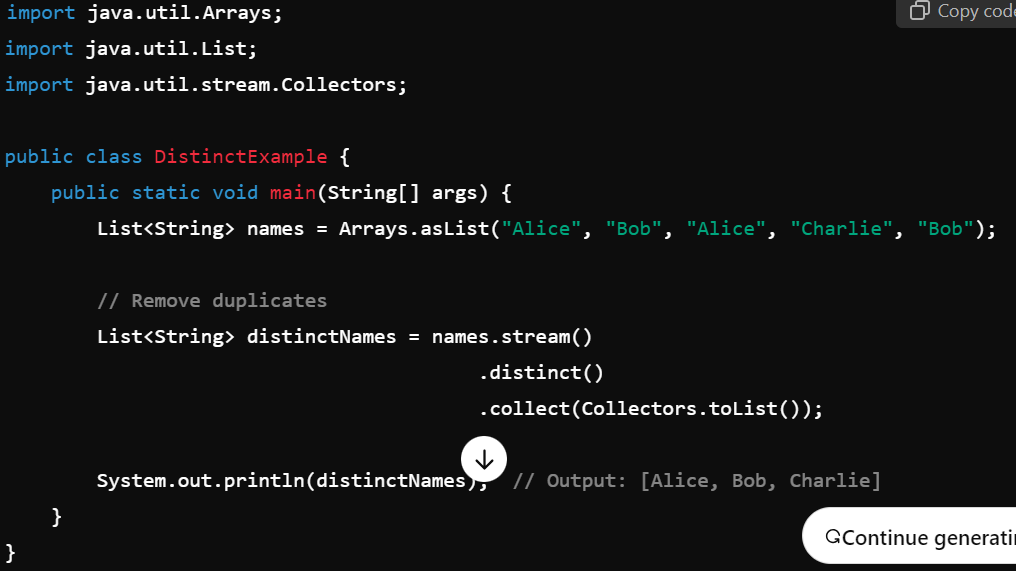
1. **flatMap() - Intermediate Operation**

Flattens a stream of collections into a single stream.



1. **distinct() - Intermediate Operation**

Removes duplicate elements from a stream.



1. **sorted() - Intermediate Operation**

Sorts the stream’s elements. You can provide a custom comparator if needed.

A screen shot of a computer program

Description automatically generated

1. **reduce() - Terminal Operation**

Reduces the elements of the stream to a single result using an associative accumulation function.

A computer screen with white text and black background

Description automatically generated

1. **collect() - Terminal Operation**

Collects the stream’s elements into a collection (e.g., List, Set).

A screenshot of a computer program

Description automatically generated

1. **forEach() - Terminal Operation**

**Performs an action for each element in the stream.**

**A screenshot of a computer program

Description automatically generated**

1. **count() - Terminal Operation**

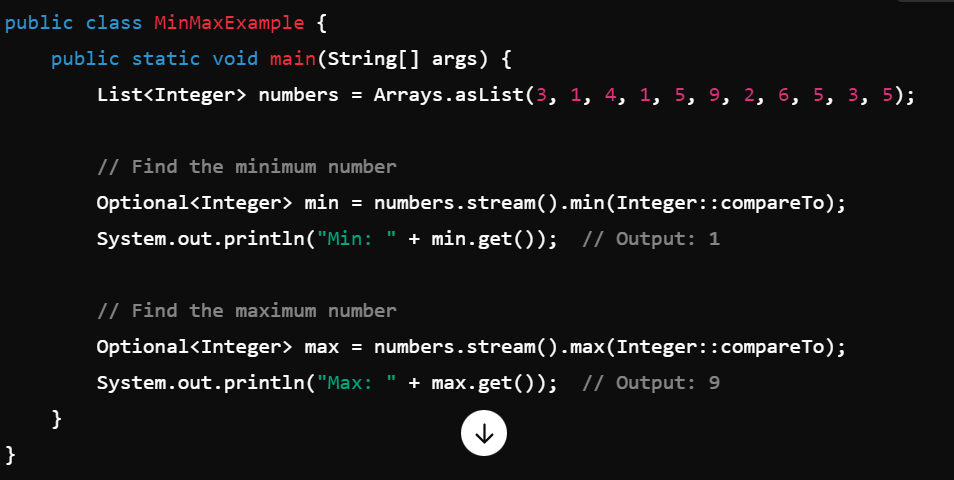
**Counts the number of elements in the stream**

**A computer screen shot of a program

Description automatically generated**

1. **min() and max() - Terminal Operations**

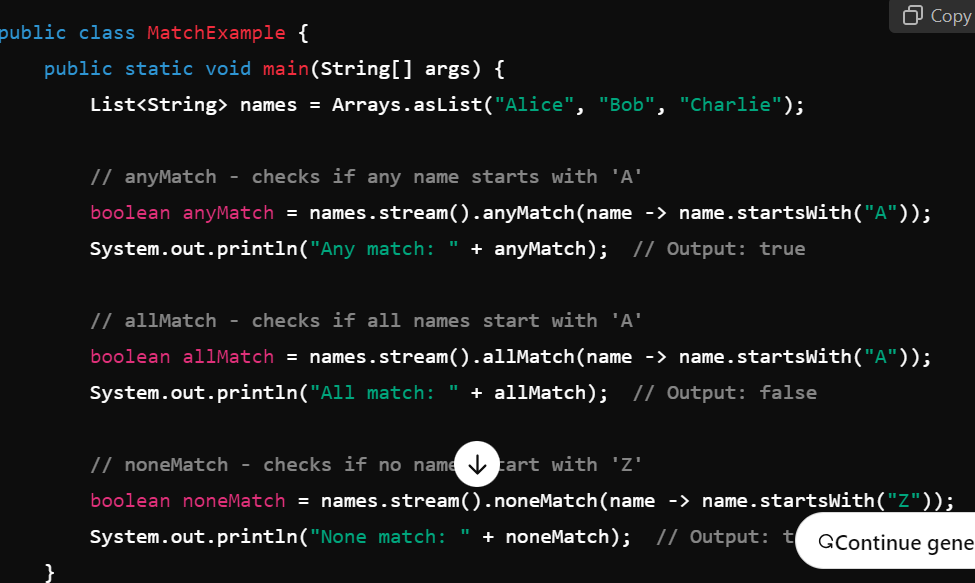
**Finds the minimum or maximum element based on a comparator.**

****

1. anyMatch(), allMatch(), noneMatch() - Terminal Operations

These are used for testing conditions on elements in the stream.

* anyMatch(): Returns true if any element matches the given predicate.
* allMatch(): Returns true if all elements match the given predicate.
* noneMatch(): Returns true if no element matches the given predicate.

****

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

Answer:

* map(): Transforms each element in the stream into another object. It returns a new stream of the transformed elements.
* flatMap(): Used when each element can be transformed into multiple elements (a collection), and it flattens the resulting stream of streams into a single stream.

6. What is the purpose of Optional in Java 8?

Answer: Optional is a container object which may or may not contain a non-null value. It helps avoid NullPointerException by explicitly indicating that a value may be absent.

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

// If present, print the value; otherwise, print a default message.

name.ifPresent(System.out::println);

System.out.println(name.orElse("Unknown"));

Key Methods:

* isPresent(): Returns true if the value is present, otherwise false.
* ifPresent(): Executes a block of code if the value is present.
* orElse(): Returns the value if present, or a default value otherwise.
* orElseThrow(): Returns the value if present, or throws an exception.

7. What is the difference between forEach() and collect() in Java 8 Streams?

Answer:

* forEach(): A terminal operation that iterates through the elements of the stream and performs an action on each element.
* collect(): A terminal operation that transforms the elements of the stream into a different form, such as a collection.



8. How does Java 8 handle default methods in interfaces?

Answer: Java 8 allows interfaces to have default methods, which can provide concrete implementations. This was introduced to ensure backward compatibility when adding new methods to existing interfaces without breaking implementing classes.

With default methods, you can now provide a **default implementation** in the interface itself, allowing classes to **inherit the method** or **override it** if needed.

A computer screen with white text

Description automatically generated

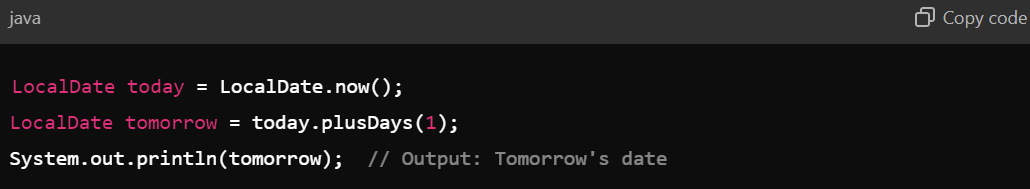
9. What is the new Date and Time API introduced in Java 8?

Answer: The new java.time package in Java 8 provides a comprehensive, immutable, and thread-safe API for handling dates, times, and time zones, replacing the older java.util.Date and java.util.Calendar.

Key Classes:

* LocalDate: Represents a date without time.
* LocalTime: Represents a time without a date.
* LocalDateTime: Represents both date and time.
* ZonedDateTime: Represents date and time with a time zone.
* Duration: Represents a time-based amount of time (e.g., seconds, minutes).
* Period: Represents a date-based amount of time (e.g., years, months).

Example:



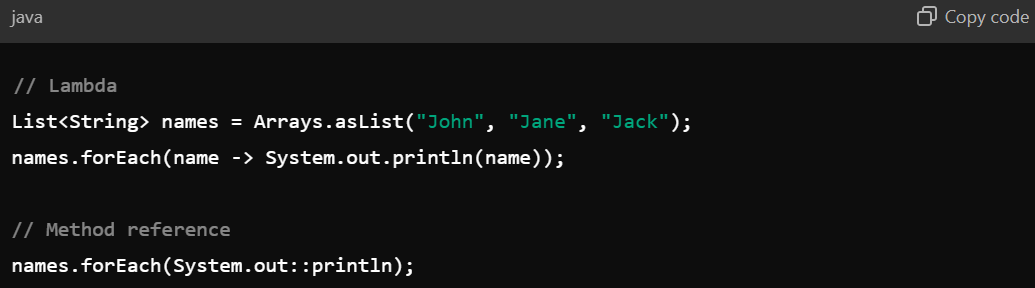
10. What are Method References in Java 8?

Answer: Method references are a shorthand notation for lambda expressions that directly refer to a method without executing it.

Types:

* Static method reference: ClassName::methodName
* Instance method reference: instance::methodName
* Constructor reference: ClassName::new

Example:



11. What is the difference between Stream and Collection?

Answer:

* Collection: It represents a collection of elements that are stored in memory and manipulated in place (e.g., List, Set). You typically use loops to iterate over collections.
* Stream: It provides a functional approach to working with data. Streams are not stored in memory but are pipelines for processing collections, and they support operations like filtering, mapping, and reducing in a more declarative way.

12. What are the advantages of using Java 8 features like Lambda and Streams?

Answer:

* Concise Code: Lambda expressions make code more readable and concise.
* Parallel Processing: Streams make it easier to process collections in parallel using parallelStream().
* Functional Style: Promotes a functional programming style, making operations more declarative.
* Less Boilerplate: Reduces the need for boilerplate code, especially for operations on collections and APIs.