**Java 8 Key Features (Most Important for Interviews):**

1. ✅ **Lambda Expressions**

**🔸 What is a Lambda Expression?**

A lambda is a short block of code that takes in parameters and returns a value.

In Java 8, Lambda expressions allow us to write simpler, more compact code. It’s a way to pass code as a parameter to a method. It looks like a function, but without the method name.

**Basic Syntax:**

(parameter) -> expression

**🌟 Key Points:**

1. **No need for a class**: In normal Java, if you want to define a function, you need to put it inside a class or an interface. With lambda, you can define it inline (on the spot).
2. **Concise code**: Lambdas help reduce the amount of code by eliminating boilerplate (extra, repetitive code).

**⚡ Example 1: Basic Lambda Expression**

Let's take a look at a simple example where we print a message:

**Without Lambda:**

interface Message {

void sayHello();

}

class MyMessage implements Message {

public void sayHello() {

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

}

}

public class Main {

public static void main(String[] args) {

Message msg = new MyMessage();

msg.sayHello();

}

}

**With Lambda:**

You can achieve the same thing with a **lambda expression** in a much shorter way:

interface Message {

void sayHello();

}

public class Main {

public static void main(String[] args) {

Message msg = () -> System.out.println("Hello, World!");

msg.sayHello();

}

}

**Example 2: Lambda with Parameters**

Let’s take an example where you want to perform a calculation. The lambda will take two parameters and add them:

interface Calculator {

int add(int a, int b);

}

public class Main {

public static void main(String[] args) {

// Lambda that adds two numbers

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

// Calling the add method

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

}

}

Here, (a, b) -> a + b is the lambda expression, where a and b are parameters and a + b is the body.

**📚 What is Lambda Expression in Java?**

➔ **Lambda Expression** = A **shortcut way** to write a method — **without a name** and **inside a body**.  
➔ It’s mainly used to **pass behavior as a parameter**.

**In short:**  
Lambda Expression = **Anonymous Method** = **Pass behavior easily**

// Without Lambda

Runnable r = new Runnable() {

public void run() {

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

}

};

// With Lambda

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

**🎯 Lambda Expression — Interview Questions + Answers**

**1. What is a Lambda Expression?**

**✅***Answer*:A lambda expression is a short way to write anonymous functions to implement interfaces (mostly functional interfaces)

**Note:** **An anonymous function is a function without a name. like ()-> ,normal functions are int add()**

**2. Why do we need Lambda Expressions in Java?**

*✅ Answer:*

* To reduce boilerplate code.
* To improve readability.
* To promote functional programming.

**3. Can a lambda expression have multiple lines of code inside it?**

**✅ Answer:** Yes. If multiple lines, use { } curly braces.

Runnable r = () -> {

System.out.println("Hello");

System.out.println("World");

};

**4. Can lambda expressions throw exceptions?**

✅ **Answer:** Yes. You can handle or throw exceptions inside lambda blocks.

**5. How is a lambda related to functional interfaces?**

**✅ Answer:** A lambda implements a functional interface by providing the definition for the abstract method.

**6. Can we create a thread using a lambda expression? How?**

✅ **Answer**: new Thread(() -> System.out.println("Thread running")).start();

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. ✅ Functional Interfaces

🧩 **Functional Interface in Java**

* A **Functional Interface** is an interface that contains exactly one **abstract method**.
* It can have multiple **default** or **static methods**, but it can have only one **abstract method**.
* Functional interfaces are used primarily to support **lambda expressions** in Java, which allows for more concise and readable code.
* The main reason we use **functional interfaces** in Java is to support **functional programming** paradigms and to enable more **flexible**, **concise**, and **maintainable** code.
* Use annotation to declare functional interface

**Example**

@FunctionalInterface

interface MathOperation {

int operate(int a, int b);

}

public class LambdaExample {

public static void main(String[] args) {

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

System.out.println(add.operate(10, 5)); // Output: 15

}

| **Feature** | **Interface** | **Functional Interface** |
| --- | --- | --- |
| Abstract Methods | Can have multiple abstract methods | Exactly one abstract method |
| Default Methods | Can have default and static methods | Can have default and static methods as well |
| Functional | Not specifically designed for functional programming | Designed for use with lambda expressions and method references |
| Annotation | Not required to use @FunctionalInterface | Can be annotated with @FunctionalInterface (optional) |
| Usage | Used in traditional object-oriented programming | Primarily used to define behavior that can be passed around (used with lambdas) |

**When to Use Each**

* **Regular Interface**: Use a regular interface when you want to define a contract with **multiple behaviours** (methods) that need to be implemented by the class.
* **Functional Interface**: Use a functional interface when you want to pass a **single behaviour** (like a **function** or **action**) as an argument to a method, or when you are utilizing **lambda expressions** for concise code.
* **Abstract Methods** are used to **define required behavior** that must be implemented by the concrete classes. They don’t have a body and are mandatory for the implementing classes.
* **Default Methods** are used to **provide optional behavior** that can be inherited directly by implementing classes without needing to implement it. They provide default functionality, making interfaces more flexible.

**🎯 Functional Interface — Interview Questions + Answers**

**1. What is a Functional Interface?**

**✅ Answer:**

An interface with exactly one abstract method is called a functional interface.

Example: Runnable, Callable, Comparator.

**2. Can a Functional Interface have default and static methods?**

**✅ Answer:** Yes. It can have multiple default/static methods, but only one abstract method

**3. What happens if a functional interface has 2 abstract methods?**

**✅ Answer:** It is no longer a functional interface and cannot be used for lambda expressions

**4. Explain @FunctionalInterface annotation. Is it mandatory?**

**✅ Answer:** It’s optional, but if you use it, the compiler will throw an error if you add more than one abstract method.

**5. Give examples of built-in functional interfaces in Java 8.**

**✅ Answer:**

* Predicate<T> — test condition and returns boolean.
* Consumer<T> — accepts a value, returns nothing.
* Supplier<T> — supplies a value.
* Function<T, R> — takes input, returns output.

**6. Can you create your own Functional Interface?**

**✅ Answer:** Yes! Example:

@FunctionalInterface

interface MyInterface {

void myMethod();

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. ✅ Stream API

**📚 What is Stream API?**

➔ Stream API is used to **process collections (like List, Set)**  
in a **very easy, fast, and functional style**.

Think of it like this:

**Collection**=store data 📦

**Stream** = process data 🔥

**✍ Basic Example:**

**Without Stream (Old style):**

List<String> names = Arrays.asList("Rahul", "Anjali", "Tanveer");

for (String name : names) {

if (name.startsWith("T")) {

System.out.println(name);

}

}

**With Stream (New style):**

List<String> names = Arrays.asList("Rahul", "Anjali", "Tanveer");

names.stream()

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

.forEach(System.out::println);

| **Feature** | **Explanation** |
| --- | --- |
| Stream is not a data structure | It's just a pipeline to **process** data. |
| Streams are lazy | They only execute when needed (performance boost). |
| Stream doesn't modify original data | It gives a **new result**. (Immutable processing) |
| Functional Programming style | You use **filter, map, reduce** functions. |

**Key Points About Stream:**

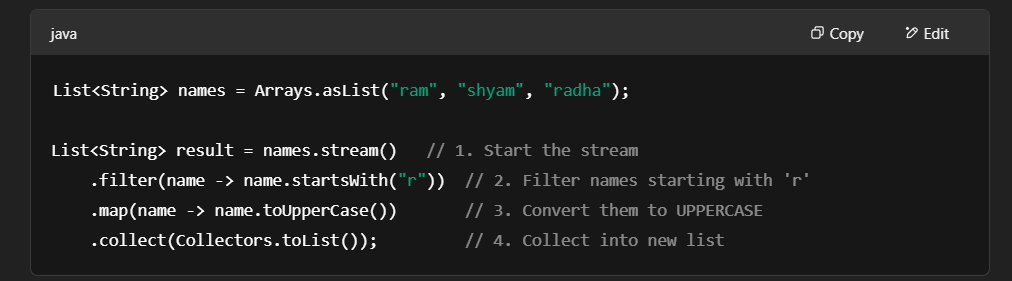
**Why do we use stream()?**

* stream() **converts a Collection (like List, Set, etc.) into a Stream**.
* Without calling .stream(), we cannot use filter(), map(), collect(), etc.

**Which method is used first?**

| **Step** | **What we call** | **Why we call it** |
| --- | --- | --- |
| 1 | .stream() | Start processing the list |
| 2 | .filter() (optional) | Select specific elements |
| 3 | .map() (optional) | Modify elements |
| 4 | .sorted() (optional) | Sort elements |
| 5 | .collect() | Gather final result |

✅ **Always this order:**

Example of flow:

✅ In one line:

➔ stream() = open pipeline

➔ then filter()/map()/sorted()/etc. = **operations**

➔ collect() = close and gather result.

🚀 Stream API: Important Methods

| **Method** | **Purpose** | **Example** | **Meaning** |
| --- | --- | --- | --- |
| stream() | Start processing elements | list.stream() | Convert List/Set into Stream |
| filter(Predicate) | Select elements based on a condition | .filter(x -> x > 10) | Keep only numbers > 10 |
| map(Function) | Transform each element | .map(x -> x \* 2) | Double each number |
| sorted() | Sort elements | .sorted() | Ascending order by default |
| sorted(Comparator) | Custom sort | .sorted((a, b) -> b - a) | Descending order |
| collect(Collector) | Gather result into List/Set/Map | .collect(Collectors.toList()) | Store in a list |
| forEach(Consumer) | Do something with each element | .forEach(System.out::println) | Print all elements |
| count() | Count elements after processing | .count() | Total elements |
| findFirst() | Get first element (Optional) | .findFirst() | Useful for search |

**1️⃣ stream()**

List<String> names = Arrays.asList("ram", "shyam", "radha");

Stream<String> stream = names.stream();

**2️⃣ filter()**

names.stream()

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

.forEach(System.out::println);

➔ Select only names starting with "r".

**3️⃣ map()**

names.stream()

.map(name -> name.toUpperCase())

.forEach(System.out::println);

➔ Change each name to UPPERCASE.

**✅ flatMap()**

* **Purpose**: 👉 **Flatten** (break and merge) multiple collections into a single stream.
* **Output**: 👉 A **flat single Stream** (no nested lists).

**Differ**

map():

[ [1,2], [3,4] ] --> [ Stream(1,2), Stream(3,4) ] --> (Nested)

flatMap():

[ [1,2], [3,4] ] --> [1,2,3,4] --> (Flattened)

**📚 What Are Collectors?**

* **Collectors** are **utility methods** used to **collect** the processed data from a **Stream** into different data structures or forms like:
  + List
  + Set
  + Map
  + Single values (sum, count, average, etc.)
  + Grouping or partitioning data
* They are part of the **java.util.stream.Collectors** class.
* Used with the **terminal operation** collect().

| **Collector** | **Use Case** | **Output Type** |
| --- | --- | --- |
| toList() | Convert stream to List | List<T> |
| toSet() | Convert stream to Set (removes duplicates) | Set<T> |
| toMap() | Create Map using key-value mapping | Map<K, V> |
| counting() | Count elements | Long |
| joining() | Concatenate strings with delimiter | String |
| groupingBy() | Group elements based on a classifier | Map<key, List<values>> |
| partitioningBy() | Split elements into two groups (true/false) | Map<Boolean, List<T>> |
| summarizingInt() | Get statistics (count, sum, min, avg, max) | IntSummaryStatistics |

**7️⃣ count()**

long total = names.stream()

.filter(name -> name.length() > 3)

.count();

System.out.println(total);

➔ Count how many names have more than 3 letters.

**8️⃣ findFirst()**

Optional<String> first = names.stream()

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

.findFirst();

System.out.println(first.get());

➔ **filter** = selection

➔ **map** = transformation

➔ **sorted** = ordering

➔ **collect** = saving

➔ **forEach** = action (like print)

➔ **count/findFirst** = search, calculation

**🎯 Stream API — Interview Questions**

**1. What is Stream API? Why was it introduced in Java 8?**

✅ **Answer:**

Stream API helps process data (like List, Set) easily using functional-style operations (filter, map, sort, etc.).  
It was introduced to reduce boilerplate code (loops), improve readability, and support parallel operations.

**2. What is the difference between Collection and Stream?**

✅ **Answer:**

* Collection stores data (data structure) — can be modified.
* Stream processes data — does not store or modify the source.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3. Is a Stream reusable once used?**

✅ **Answer:**

No, streams are like water flow.

Once operated, they are **consumed** and **cannot be reused**.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**4. Difference between map() and flatMap()?**

✅ **Answer:**

* map() → transforms each element individually.
* flatMap() → transforms and flattens nested structures into a single stream.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**5. What is the difference between filter() and map()?**

✅ **Answer:**

* filter() → keeps only elements that match a condition.
* map() → changes each element to something else

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**6. What happens if you don't use terminal operation?**

✅ **Answer:**

Nothing happens! Streams are **lazy**.

Intermediate operations like map(), filter() won't execute until a **terminal operation** like collect(), forEach(), count() is called.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**7. Explain lazy evaluation in Stream API.**

✅ **Answer:**

Operations in streams (like filter, map) are not executed immediately.  
They are only evaluated when a terminal operation is invoked.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**8. What is the purpose of collect() and what are Collectors?**

✅ **Answer:**

* collect() gathers the processed stream elements into a Collection (List, Set, Map).
* Collectors is a utility class having ready-made methods like toList(), toSet(), joining(), etc.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**9. Difference between parallelStream() and stream()?**

✅ **Answer:**

* stream() → sequential processing (one after another).
* parallelStream() → processing in multiple threads (parallel).

1. ✅ Method References

**What is a Method Reference?**

A **Method Reference** is a shorthand notation of a lambda expression to call a method.

It uses the **::** operator to refer to a method without invoking it.

**Types of Method References:**

1. **Static Method Reference**: Refers to a static method in a class.

**1. Static Method Reference:**

Static methods are referenced using the class name.

**Syntax:**

ClassName::staticMethod

**Example:**

import java.util.Arrays;

import java.util.List;

public class **StaticMethodReference** {

// Static method

public static void printUpperCase(String str) {

System.out.println(str.toUpperCase());

}

public static void main(String[] args) {

List<String> names = Arrays.asList("Java", "Python", "JavaScript");

// Using Method Reference to call static method

names.forEach(**StaticMethodReference::printUpperCase**);

}

}

1. **Instance Method Reference**: Refers to an instance method of a class.

**Syntax:**

object::instanceMethod

**Example:**

import java.util.Arrays;

import java.util.List;

public class InstanceMethodReference {

// Instance method

public void printLowerCase(String str) {

System.out.println(str.toLowerCase());

}

public static void main(String[] args) {

List<String> names = Arrays.asList("Java", "Python", "JavaScript");

InstanceMethodReference instance = new InstanceMethodReference();

// Using Method Reference to call instance method

names.forEach(instance::printLowerCase);

}

}

1. **Constructor Method Reference**: Refers to the constructor of a class.
2. **Referring to an Instance Method of an Arbitrary Object**: Refers to an instance method of an object.

import java.util.Arrays;

import java.util.List;

public class ArbitraryObjectMethodReference {

public static void main(String[] args) {

List<String> names = Arrays.asList("Java", "Python", "JavaScript");

// Using Method Reference to call instance method on an arbitrary object

names.forEach(String::toUpperCase); // Using method reference for String class

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. ✅ Default & Static Methods in Interfaces

**1. Default Methods in Interfaces**

In Java 8, interfaces can now have **default methods**. A default method is a method that has an implementation inside the interface. This was introduced to allow developers to add new methods to interfaces without breaking the existing implementations of the interface.

interface Animal {

default void sound() {

System.out.println("Animal makes a sound");

}

void eat(); // Abstract method

}

class Dog implements Animal {

@Override

public void eat() {

System.out.println("Dog is eating");

}

// Optionally, the sound() method can be overridden if needed

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Calls the default method from the interface

dog.eat(); // Calls the implemented eat() method in Dog

}

}

**2. Static Methods in Interfaces**

In Java 8, interfaces can also have **static methods**. Static methods in interfaces are similar to static methods in regular classes, but they are part of the interface and are not inherited by implementing classes.

**Key Points About Static Methods:**

* **Can’t Be Overridden**: Static methods cannot be overridden by implementing classes.
* **Called on the Interface**: Static methods are invoked on the interface itself, not on instances of the implementing classes.
* **Cannot be accessed through instances** of the implementing class (they are accessed through the interface).

interface Vehicle {

static void info() {

System.out.println("This is a static method in the Vehicle interface");

}

void drive();

}

class Car implements Vehicle {

@Override

public void drive() {

System.out.println("Car is driving");

}

}

public class Main {

public static void main(String[] args) {

// Call the static method on the interface, not the class

Vehicle.info();

Car car = new Car();

car.drive();

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. ✅ Optional Class

**What is Optional?**

Optional is a container object introduced in Java 8 that can hold either:

* A non-null value, or
* Nothing (i.e., null).
* It helps you to avoid NullPointerExceptions by providing a clear and safe way to handle missing values.

**Why Use Optional?**

The main purpose of **Optional** is to represent the idea **that a value may or may not be present**. It allows you to write more readable and cleaner code, and handle cases where values might be missing without directly using null.

**When to Use Optional**

* **Use Optional for return types**: It's great when a method might return a value or might not (like null).
* **Avoid using Optional for fields**: It's not meant to be used for object fields, as it's more intended for method return types.
* **Avoid using Optional in collections**: If you're working with collections, don't wrap every item in an Optional.

**Summary**

* Optional is a container that either contains a value or is empty.
* It provides methods like isPresent(), ifPresent(), orElse(), and map() to safely handle values that might be missing.
* Optional helps prevent NullPointerException and makes your code more readable and expressive.

1. ✅ Date and Time API (java.time package)

**Date and Time API** introduced in Java 8 as part of the java.time package.

This new API was introduced to address the limitations and confusion caused by the old java.util.Date and java.util.Calendar classes, offering a more modern and comprehensive solution for handling dates and times.

**Overview of java.time Package**

The **java.time** package provides a set of classes to handle dates, times, durations, and time zones with more flexibility and less confusion. The key benefits are:

* **Immutability**: All classes in java.time are immutable, which means their state can't be changed once they are created.
* **Thread Safety**: Since they are immutable, they are thread-safe.
* **Clear Separation**: The API clearly separates the concepts of date, time, and time zones.
* **Better API Design**: More expressive and easy-to-use methods for manipulating and formatting date-time values.

**Key Classes in java.time API**

1. **LocalDate**: Represents a date (year, month, day) without the time of day or timezone.
2. **LocalTime**: Represents a time (hour, minute, second, and nanosecond) without a date or timezone.
3. **LocalDateTime**: Combines both a date and a time without a timezone.
4. **ZonedDateTime**: Represents a full date-time with timezone information.
5. **Instant**: Represents a point in time, typically measured from the Unix epoch (January 1, 1970).
6. **Duration**: Represents the amount of time between two Temporal objects (e.g., seconds, minutes).
7. **Period**: Represents the amount of time in terms of years, months, and days.

| **Feature** | **java.time API** | **java.util (Date/Calendar)** |
| --- | --- | --- |
| **Immutability** | All classes are immutable | Mutable classes (Date, Calendar) |
| **Thread Safety** | Thread-safe due to  immutability | Not thread-safe |
| **Date and Time**  **Separation** | Clear separation between date, time, and timezone | No clear separation, often combined |
| **Ease of Use** | Intuitive and easy to use | Difficult to use and error-prone |
| **Time Zone Support** | Direct support for time zones  (ZonedDateTime) | Limited time zone support |
| **Formatting and**  **Parsing** | Uses DateTimeFormatter  (flexible) | Uses SimpleDateFormat (less flexible) |
| **Performance** | Better performance, especially with multi-threading | Less efficient, especially with multi-threading |
| **API Evolution** | Modern API designed for current use cases | Legacy API, less suited for modern needs |
| **Conversion to/from java.time** | Simple and built-in conversion | Conversion requires additional effort |

1. ✅ Collectors & Terminal Operations

A **Collector** is a special type of **reducer** in the context of the Java Streams API. It is used to accumulate the elements of a stream into a **collection**, like a List, Set, or Map, or perform some other reduction operation (like counting elements or summing values). Collectors are used as the **final step** of a stream pipeline.

A **Terminal Operation** is the final operation that triggers the processing of a stream. Once a terminal operation is applied, the stream is considered consumed and can no longer be used.

**New java.util Collection Features**

* **Purpose**: Java 8 added several methods to the collections framework to make working with collections easier.
* **Key Concepts**:
  + **forEach**() method in Iterable
  + **removeIf**() method for removing elements based on a condition
  + **replaceAll**() for modifying elements
  + **computeIfAbsent**() and **computeIfPresent**() in Map
  + Stream support for collections