JAVA 8+

Interview Questions & Answers

A Complete, Systematic Guide with Theory, Explanations & Code Examples

Compiled & Published by

SkilltoGrowth

9 Core Topics | 29+ In-Depth Q&A | Real Code Examples

About This Guide

This eBook is designed to give you genuine value for your time: a systematic, interview-ready walkthrough of the Java 8+ features that transformed how modern Java code is written — a heavily tested area in interviews at every level, from junior developer screens to senior backend engineering rounds. Rather than offering one-line definitions, each answer combines clear theory with practical reasoning and runnable code examples, so you understand the *why* behind every answer, not just the *what*.

Each of the 9 topics below mirrors a core area interviewers consistently probe: the Stream API, functional interfaces, lambda expressions, method references, Optional, default and static interface methods, the modern Date & Time API, and CompletableFuture for asynchronous programming. Every question is followed by a detailed explanation and, where relevant, a code snippet with its expected output — so you can study the logic and verify your understanding at the same time.

How to Use This Guide

1. Read each topic introduction first to anchor the concept before diving into individual questions.
2. Try to answer each question yourself before reading the provided answer — active recall builds stronger retention than passive reading.
3. Type out and run the code examples wherever possible; modifying them slightly is one of the fastest ways to deepen real understanding.
4. Revisit the Table of Contents to jump directly to topics you find weaker before an interview.

Disclaimer: *This guide is intended for educational and interview-preparation purposes. Code examples are simplified to illustrate core concepts clearly.*

Table of Contents

1. Stream API

The Stream API, introduced in Java 8, provides a declarative way to process sequences of elements using functional-style operations such as filter, map, and reduce. It is one of the most transformative additions to modern Java and a near-guaranteed interview topic.

Q1. What is a Stream, and how is it different from a Collection?

A Stream is not a data structure that stores elements; instead, it is a sequence of elements that supports pipelined, functional-style operations computed on demand from a source (a Collection, array, I/O channel, or generator function).

Key differences from a Collection: a Collection is about storing and efficiently accessing elements, while a Stream is about describing computations to be performed on elements. Streams are lazily evaluated (nothing happens until a terminal operation is invoked), can be traversed only once (a Stream cannot be reused after a terminal operation consumes it - attempting to reuse it throws IllegalStateException), and do not modify their underlying source.

CODE EXAMPLE

```
List<String> names = List.of("Alice", "Bob", "Charlie", "Dave");
List<String> result = names.stream()
    .filter(n -> n.length() > 3)
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());
System.out.println(result);
```

OUTPUT

```
[ALICE, CHARLIE, DAVE]
```

Q2. What is the difference between intermediate and terminal operations, and why does laziness matter?

Intermediate operations (filter(), map(), sorted(), distinct(), limit(), etc.) return a new Stream, allowing operations to be chained together. Crucially, they are lazy - they don't actually execute any processing when called; they simply build up a description of the pipeline.

Terminal operations (collect(), forEach(), reduce(), count(), anyMatch(), etc.) trigger the actual traversal of the stream and produce a final, non-stream result (a value, collection, or side effect). Only when a terminal operation is invoked does the entire pipeline actually execute, element by element, in a single pass.

This laziness matters because it enables important optimizations, such as short-circuiting (a limit(3) combined with an infinite stream will only ever process as many elements as needed to satisfy the pipeline) and fusing multiple intermediate operations into a single pass over the data rather than materializing an intermediate collection after each step.

CODE EXAMPLE

```
Stream<Integer> stream = Stream.of(1, 2, 3).map(x -> {
    System.out.println("Mapping " + x);
    return x * 2;
});
System.out.println("Stream created, nothing printed yet");
stream.forEach(System.out::println); // NOW the pipeline actually runs
```

OUTPUT

```
Stream created, nothing printed yet
Mapping 1
2
Mapping 2
4
Mapping 3
6
```

Q3. Explain the difference between map() and flatMap().

map(Function) transforms each element of a stream into exactly one new element, producing a stream of the same size (a one-to-one transformation). It is useful when each input naturally corresponds to a single output.

flatMap(Function) transforms each element into a stream of zero or more elements, and then "flattens" all those individual streams into a single, combined stream (a one-to-many transformation, followed by flattening). It is essential when working with nested structures, such as a list of lists, where you want a single flat stream of all inner elements.

CODE EXAMPLE

```
List<List<Integer>> nested = List.of(List.of(1, 2), List.of(3, 4), List.of(5));

// map() would produce a Stream<List<Integer>> - still nested
// flatMap() flattens it into a single Stream<Integer>
List<Integer> flat = nested.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
System.out.println(flat);
```

OUTPUT

```
[1, 2, 3, 4, 5]
```

Q4. How does reduce() work, and what is a practical example?

reduce() is a terminal operation that combines all elements of a stream into a single result by repeatedly applying a BinaryOperator (an accumulator function). The common three-argument overload, reduce(identity, accumulator, combiner), takes a starting value (identity), a function to combine the running result with each element, and a combiner function used to merge partial results when the stream is processed in parallel.

CODE EXAMPLE

```
List<Integer> nums = List.of(1, 2, 3, 4, 5);
int sum = nums.stream().reduce(0, Integer::sum);
int product = nums.stream().reduce(1, (a, b) -> a * b);
System.out.println(sum);
System.out.println(product);
```

OUTPUT

```
15
120
```

Q5. What is the difference between Collectors.toList() and using Collectors.groupingBy()?

Collectors.toList() is a simple collector that accumulates stream elements into a new List, preserving encounter order.

Collectors.groupingBy(classifier) is a much more powerful collector that partitions elements into a Map<K, List<T>>, grouping elements by a key derived from a classifier function - conceptually similar to a SQL GROUP BY clause. It can be combined with a downstream collector (a second argument) to further transform each group, such as counting members or computing an average, rather than just collecting raw elements into lists.

CODE EXAMPLE

```
List<String> words = List.of("apple", "banana", "avocado", "blueberry", "cherry");
Map<Character, List<String>> grouped = words.stream()
    .collect(Collectors.groupingBy(w -> w.charAt(0)));
System.out.println(grouped);

Map<Character, Long> counts = words.stream()
    .collect(Collectors.groupingBy(w -> w.charAt(0), Collectors.counting()));
System.out.println(counts);
```

OUTPUT

```
{a=[apple, avocado], b=[banana, blueberry], c=[cherry]}
{a=2, b=2, c=1}
```

Q6. What is the difference between a sequential and a parallel stream, and what are the risks of using parallel streams carelessly?

A sequential stream processes elements one after another on a single thread, in encounter order. A parallel stream (created via parallelStream() or .parallel()) splits the source data into multiple chunks processed concurrently across threads from the common ForkJoinPool, then combines the partial results.

Risks include: using non-thread-safe mutable state inside a lambda passed to a parallel stream (e.g., incrementing a shared, non-atomic counter, which causes race conditions), overhead from splitting/merging work outweighing the benefit for small datasets or cheap operations (making parallel streams actually slower), and order-sensitive operations (like forEach on an ordered stream) potentially executing out of order unless forEachOrdered() is used instead. Parallel streams should be reserved for large datasets with computationally expensive, independent, stateless operations.

2. Functional Interface

A functional interface is an interface with exactly one abstract method, serving as the target type for lambda expressions and method references. Understanding functional interfaces is prerequisite knowledge for everything else in Java 8's functional programming toolkit.

Q1. What is a functional interface, and what does @FunctionalInterface do?

A functional interface is any interface that contains exactly one abstract method (referred to as its Single Abstract Method, or SAM). It may also contain any number of default and static methods (which have implementations and do not count toward the single-abstract-method rule) as well as methods inherited from Object (like equals() or toString(), which also don't count).

@FunctionalInterface is an optional but recommended annotation that instructs the compiler to verify the interface truly has exactly one abstract method, causing a compile-time error if a second abstract method is accidentally added later. It serves purely as a documentation and safety aid - a lambda can still target an interface that qualifies as functional even without this annotation present.

CODE EXAMPLE

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b); // the single abstract method
    // default and static methods are allowed without breaking the rule:
    default void printInfo() { System.out.println("A calculator"); }
}

Calculator add = (a, b) -> a + b;
System.out.println(add.calculate(3, 4));
```

OUTPUT

```
7
```

Q2. What are the four core functional interfaces in java.util.function, and what does each represent?

Function<T, R> - takes an argument of type T and returns a result of type R, via the method apply(T). Represents a transformation.

Predicate<T> - takes an argument of type T and returns a boolean, via test(T). Represents a condition or filter.

Consumer<T> - takes an argument of type T and returns nothing (void), via accept(T). Represents an action performed on a value.

Supplier<T> - takes no arguments and returns a value of type T, via get(). Represents a factory or lazy value provider.

There are also two-argument variants (BiFunction, BiPredicate, BiConsumer) and primitive-specialized variants (IntFunction, ToIntFunction, IntPredicate, etc.) to avoid unnecessary autoboxing.

CODE EXAMPLE

```
Function<Integer, Integer> square = x -> x * x;
Predicate<Integer> isEven = x -> x % 2 == 0;
Consumer<String> printer = s -> System.out.println("Got: " + s);
Supplier<String> greeting = () -> "Hello!";

System.out.println(square.apply(5));
System.out.println(isEven.test(4));
printer.accept("test");
System.out.println(greeting.get());
```

OUTPUT

```
25
true
Got: test
Hello!
```

Q3. How can you combine multiple functional interfaces using default methods like andThen() and compose()?

Several built-in functional interfaces provide default methods for chaining/combining behavior. Function.andThen(after) runs the current function first, then feeds its result into after; Function.compose(before) runs before first, then feeds its result into the current function - the two differ in the order of execution. Predicate offers and(), or(), and negate() for combining boolean conditions logically.

CODE EXAMPLE

```
Function<Integer, Integer> times2 = x -> x * 2;
Function<Integer, Integer> plus3 = x -> x + 3;

System.out.println(times2.andThen(plus3).apply(5)); // (5*2)+3 = 13
System.out.println(times2.compose(plus3).apply(5)); // (5+3)*2 = 16

Predicate<Integer> isPositive = x -> x > 0;
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isPositive.and(isEven).test(4)); // true
```

OUTPUT

```
13
16
true
```

3. Lambda Expression

Lambda expressions provide a concise way to represent an instance of a functional interface, enabling a functional programming style within Java. They are the syntax that makes the Stream API and functional interfaces practical to use in everyday code.

Q1. What is a lambda expression, and what problem does it solve?

A lambda expression is a short block of code that takes parameters and returns a value, essentially an anonymous function that can be treated as data - passed as an argument, stored in a variable, or returned from a method. Syntactically, it takes the form (parameters) -> expression or (parameters) -> { statements; }.

Before Java 8, representing a small piece of behavior (like a comparison rule or an event handler) required writing a verbose anonymous inner class implementing a single-method interface. Lambda expressions solve this by letting you express that same behavior far more concisely, eliminating boilerplate and making functional-style operations (like those in the Stream API) practical and readable.

CODE EXAMPLE

```
// Before Java 8 - anonymous inner class
Comparator<String> byLength = new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
};

// Java 8+ - lambda expression
Comparator<String> byLengthLambda = (a, b) -> a.length() - b.length();

List<String> words = new ArrayList<>(List.of("banana", "kiwi", "apple"));
words.sort(byLengthLambda);
System.out.println(words);
```

OUTPUT

```
[kiwi, apple, banana]
```

Q2. What is variable capture in lambda expressions, and why must captured variables be effectively final?

A lambda expression can reference local variables from its enclosing scope, a behavior known as "capturing" those variables. However, any local variable captured this way must be final or "effectively final" (never reassigned after initialization, even without the final keyword).

This restriction exists because lambdas may outlive the method call that created them (e.g., a lambda passed to a thread or stored and invoked later). Local variables live on the stack and are destroyed when the method returns, so the lambda instead captures a copy of the variable's value at creation time. If the original variable could still change afterward, the lambda's captured copy and the "live" variable would silently diverge, creating a subtle and confusing class of bugs - so Java disallows this scenario entirely at compile time by enforcing effective finality.

CODE EXAMPLE

```
int counter = 0;
Runnable r = () -> System.out.println(counter); // captures 'counter'
// counter++; // if uncommented, this line causes a COMPILE ERROR:
// "local variables referenced from a lambda expression must be
//  final or effectively final"
```

Q3. What is the difference between a lambda expression and an anonymous inner class beyond syntax?

Even though lambdas can often replace anonymous inner classes implementing a single-method interface, they are not fully equivalent. A lambda does NOT create a new scope for this - inside a lambda, this refers to the enclosing class instance, whereas inside an anonymous inner class, this refers to the anonymous class instance itself.

Additionally, lambdas are compiled more efficiently using the invokedynamic bytecode instruction and the LambdaMetafactory (rather than generating a separate .class file per lambda, as anonymous inner classes do), and a lambda cannot introduce new instance fields or additional methods the way an anonymous class body can.

4. Method Reference

Method references provide an even more concise alternative to lambda expressions when a lambda would simply call an existing method. They improve readability by letting code point directly at the method being invoked.

Q1. What is a method reference, and what are its four types?

A method reference is shorthand syntax (using ::) for a lambda expression that does nothing more than call an existing method. There are four categories: 1) Reference to a static method - ClassName::staticMethod. 2) Reference to an instance method of a particular object - instance::instanceMethod. 3) Reference to an instance method of an arbitrary object of a particular type - ClassName::instanceMethod (the first lambda parameter becomes the object the method is called on). 4) Reference to a constructor - ClassName::new.

CODE EXAMPLE

```
// 1. Static method reference
Function<String, Integer> parse = Integer::parseInt;

// 2. Instance method of a particular object
String greeting = "Hello";
Supplier<Integer> lengthOf = greeting::length;

// 3. Instance method of an arbitrary object of a particular type
Function<String, Integer> lengthFn = String::length;

// 4. Constructor reference
Supplier<ArrayList<String>> listMaker = ArrayList::new;

System.out.println(parse.apply("42"));
System.out.println(lengthOf.get());
System.out.println(lengthFn.apply("Java"));
```

OUTPUT

```
42
5
4
```

Q2. How does the compiler know which type of method reference is being used, and how does it relate to the target functional interface?

The compiler determines the method reference type through context: it examines the target functional interface's single abstract method signature (number and types of parameters, return type) and matches it against candidate methods with that name. For the 'arbitrary object of a particular type' form, the compiler recognizes that the functional interface's first parameter should serve as the implicit receiver (the object the instance method is called on), with any remaining parameters passed as the method's actual arguments.

This is why String::compareTo can serve as a valid Comparator<String> (whose compare(String, String) method takes two arguments) - the compiler recognizes it should treat the first argument as the receiver (a.compareTo(b)) rather than trying to pass both as explicit arguments to a static-style call.

CODE EXAMPLE

```
Comparator<String> comp = String::compareTo;
// equivalent to: (a, b) -> a.compareTo(b)
System.out.println(comp.compare("apple", "banana"));
```

OUTPUT

```
-1  (apple sorts before banana)
```

5. Optional

Optional<T> is a container object introduced in Java 8 to represent the presence or absence of a value explicitly, offering a safer alternative to returning null and reducing NullPointerException risk.

Q1. What is Optional, and what problem was it designed to solve?

Optional<T> is a container object that may or may not hold a non-null value. It was designed primarily as a return type for methods that might not have a result to return (e.g., a database lookup that might find nothing), making the possibility of "no value" explicit in the method's signature and forcing callers to consciously handle that case, rather than silently returning null and risking an unexpected NullPointerException somewhere downstream.

It shifts null-handling from an implicit, easy-to-forget runtime risk into an explicit part of the type system that the compiler and API encourage you to address.

CODE EXAMPLE

```
Optional<String> maybeValue = Optional.of("Hello");
Optional<String> empty = Optional.empty();

System.out.println(maybeValue.isPresent()); // true
System.out.println(empty.isPresent());       // false
System.out.println(maybeValue.orElse("Default"));
System.out.println(empty.orElse("Default"));
```

OUTPUT

```
true
false
Hello
Default
```

Q2. What is the difference between orElse(), orElseGet(), and orElseThrow()?

orElse(defaultValue) returns the contained value if present, or the given default otherwise. Important gotcha: the default value expression is always evaluated eagerly, even when the Optional is present and the default won't actually be used - this can cause unnecessary work if the default is expensive to compute.

orElseGet(Supplier) returns the contained value if present, or lazily invokes the supplied Supplier to compute a default only when needed - avoiding unnecessary computation, and preferred whenever the fallback is non-trivial to produce.

orElseThrow() (no-arg, Java 10+) throws NoSuchElementException if empty; the overload orElseThrow(Supplier<Exception>) throws a custom exception instead, useful for converting an absent value into a meaningful domain-specific error.

CODE EXAMPLE

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

// orElseGet is lazy - the supplier only runs if truly needed
String value = empty.orElseGet(() -> {
    System.out.println("Computing expensive default...");
    return "Computed Default";
});
System.out.println(value);
```

OUTPUT

```
Computing expensive default...
Computed Default
```

Q3. What are some Optional anti-patterns to avoid?

Calling get() without first checking isPresent() defeats the entire purpose of Optional, since it can throw NoSuchElementException just as carelessly as a raw null reference could throw NullPointerException - prefer orElse()/orElseGet()/ifPresent()/map() instead.

Using Optional as a field type in a class or as a method parameter type is discouraged - Optional is intended specifically as a return type to signal "this method might not return a value," and using it for fields adds serialization complications and unnecessary wrapping overhead without a matching benefit.

Wrapping a collection in an Optional<List<T>> is generally unnecessary - an empty collection (Collections.emptyList()) already unambiguously represents "no results" without needing an extra layer of wrapping.

6. Default Methods

Default methods allow interfaces to provide a method implementation directly, a capability introduced in Java 8 primarily to support evolving existing interfaces (like adding forEach() to Iterable) without breaking every class that already implements them.

Q1. What is a default method, and why was it introduced?

A default method is a method in an interface that has a body, declared using the default keyword. Unlike traditional abstract interface methods, implementing classes are not required to override it - they inherit the default implementation automatically, though they may override it if desired.

It was introduced primarily to solve the "interface evolution" problem: before Java 8, adding a new method to an existing, widely-implemented interface (like Collection or Iterable) would break every existing class that implemented that interface, since they would suddenly fail to compile without providing an implementation for the new method. Default methods let the JDK team add new methods (like Iterable.forEach() or Collection.stream()) to existing interfaces while remaining fully backward-compatible with all pre-existing implementations.

CODE EXAMPLE

```
interface Vehicle {
    void drive();

    default void honk() { // new capability added without breaking existing code
        System.out.println("Beep beep!");
    }
}

class Car implements Vehicle {
    public void drive() { System.out.println("Driving..."); }
    // honk() is inherited automatically - no need to implement it
}

new Car().honk();
```

OUTPUT

```
Beep beep!
```

Q2. What happens when a class implements two interfaces that both provide conflicting default methods with the same signature?

This is a variant of the "diamond problem" discussed in the OOP guide's Inheritance section. Java does not automatically pick one default implementation over the other; instead, the compiler forces the implementing class to explicitly override the conflicting method and resolve the ambiguity itself - it will not compile otherwise. Inside that override, the class can still access either parent's specific default implementation using InterfaceName.super.methodName().

CODE EXAMPLE

```
interface A { default void greet() { System.out.println("Hello from A"); } }
interface B { default void greet() { System.out.println("Hello from B"); } }

class C implements A, B {
    @Override
    public void greet() {
        A.super.greet(); // explicitly choose A's version
        B.super.greet(); // or explicitly call B's version too
    }
}

new C().greet();
```

OUTPUT

```
Hello from A
Hello from B
```

Q3. Does adding a default method break the definition of a functional interface?

No. Default methods (and static methods) do not count toward an interface's abstract method count, so an interface with one abstract method plus any number of default methods still qualifies as a valid functional interface and can be the target of a lambda expression. This was in fact an intentional design goal, since default methods on functional interfaces (like Function.andThen() and Predicate.and()) are exactly what enables the composable, chainable style seen throughout java.util.function.

7. Static Interface Methods

Java 8 also allowed interfaces to declare static methods, giving interfaces a clean home for utility and factory logic that is closely related to the interface but doesn't belong to any specific instance.

Q1. What are static interface methods, and how do they differ from default methods?

A static method in an interface is declared with the static keyword and belongs to the interface itself, not to any implementing instance. It is called directly on the interface, like InterfaceName.methodName(), and cannot be overridden by implementing classes (it is not inherited by them at all).

This differs from a default method, which IS inherited by implementing classes, is invoked on an instance (instance.methodName()), and CAN be overridden. Static interface methods are best understood as a natural home for utility/helper logic tightly related to the interface's purpose, analogous to static methods on a regular utility class.

CODE EXAMPLE

```
interface MathOperation {
    int apply(int a, int b);

    static MathOperation add() { // static factory method
        return (a, b) -> a + b;
    }
    static MathOperation multiply() {
        return (a, b) -> a * b;
    }
}

System.out.println(MathOperation.add().apply(3, 4));
System.out.println(MathOperation.multiply().apply(3, 4));
```

OUTPUT

```
7
12
```

Q2. Where are static interface methods commonly used in the standard JDK?

The JDK uses static interface methods extensively for factory and utility logic: Comparator.comparing(), Comparator.naturalOrder(), and Comparator.reverseOrder() construct commonly needed Comparator instances; List.of(), Set.of(), and Map.of() (Java 9+) create immutable collection instances directly from the interface; and Stream.of() and Stream.empty() construct Stream instances. Grouping these factory methods directly on the relevant interface keeps related functionality logically co-located and discoverable, rather than scattering it across separate utility classes.

8. Date & Time API

Java 8 introduced java.time, a completely redesigned Date and Time API (based on the Joda-Time library) to replace the old, notoriously flawed java.util.Date and java.util.Calendar classes. It is a common interview topic testing awareness of immutability and modeling best practices.

Q1. Why was the old java.util.Date/Calendar API replaced, and what were its main problems?

The legacy API had several well-known design flaws: java.util.Date was mutable, meaning a Date object passed to another method could be modified unexpectedly, causing subtle bugs; it was also not thread-safe, making it dangerous to share across threads without external synchronization. Its API design was also confusing and error-prone - for example, years were counted from 1900 and months were zero-indexed (January was 0), leading to frequent off-by-one mistakes.

java.util.Calendar was similarly mutable and thread-unsafe, and its API was widely considered unintuitive and inconsistent. The new java.time package (JSR-310) fixes all of these issues by making its core classes immutable and thread-safe, with a clear, fluent, and well-documented API.

Q2. What are the core classes in java.time, and what does each represent?

LocalDate - represents a date without time or time zone (e.g., 2026-07-03).

LocalTime - represents a time without date or time zone (e.g., 14:30:00).

LocalDateTime - combines LocalDate and LocalTime, still without any time zone information.

ZonedDateTime - a LocalDateTime plus a specific time zone (ZoneId), fully accounting for offsets and daylight saving rules.

Instant - represents a single, precise point on the timeline in UTC (essentially a machine timestamp), commonly used for logging and recording when events occurred.

Duration - represents a time-based amount (e.g., "25 minutes"), used for measuring elapsed time between two Instant or time-based objects.

Period - represents a date-based amount (e.g., "2 years, 3 months, 5 days"), used for measuring elapsed time between two dates.

CODE EXAMPLE

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1995, Month.JUNE, 15);
Period age = Period.between(birthday, today);
System.out.println(age.getYears() + " years, " + age.getMonths() + " months");

LocalDateTime meeting = LocalDateTime.of(2026, 7, 3, 14, 30);
LocalDateTime later = meeting.plusHours(2).plusMinutes(15);
System.out.println(later);
```

OUTPUT

```
31 years, 0 months  (example - depends on today's date)
2026-07-03T16:45
```

Q3. How does java.time guarantee immutability, and why does that matter?

Every core class in java.time (LocalDate, LocalTime, LocalDateTime, Duration, Period, etc.) is immutable - any "modification" method (like plusDays(), minusHours(), withYear()) does not alter the original object; instead, it returns a brand new instance representing the updated value, leaving the original completely unchanged.

This matters for the same reasons immutability matters generally: thread safety without synchronization (multiple threads can safely share a LocalDate instance since it can never change), and elimination of an entire class of bugs caused by unexpectedly shared, mutable date objects - a very common source of subtle defects with the old java.util.Date API.

CODE EXAMPLE

```
LocalDate date = LocalDate.of(2026, 1, 1);
LocalDate nextWeek = date.plusWeeks(1); // returns a NEW object
System.out.println(date);      // unchanged: 2026-01-01
System.out.println(nextWeek);  // 2026-01-08
```

OUTPUT

```
2026-01-01
2026-01-08
```

9. CompletableFuture

CompletableFuture, introduced in Java 8, is a powerful class for writing asynchronous, non-blocking code, extending the older, more limited Future interface with composability and a rich functional API.

Q1. What is CompletableFuture, and how does it improve upon the older Future interface?

CompletableFuture<T> implements both Future<T> and CompletionStage<T>, representing a computation that will eventually produce a result (or fail with an exception) asynchronously.

The plain Future interface (since Java 5) has significant limitations: there is no way to manually complete a Future from outside, no way to chain a follow-up action to run automatically once it completes, no way to combine multiple Futures together, and its only way to retrieve the result, get(), blocks the calling thread. CompletableFuture solves all of these: it can be manually completed via complete(), it supports chaining callbacks (thenApply(), thenAccept(), etc.) that run automatically on completion, it can combine multiple futures (thenCombine(), allOf()), and it supports non-blocking callback-style consumption of results, avoiding the need to block with get().

CODE EXAMPLE

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    // simulate a long-running computation on a background thread
    return 42;
});

future.thenApply(result -> result * 2)
      .thenAccept(result -> System.out.println("Result: " + result));

// main thread is NOT blocked while the computation runs
```

OUTPUT

```
Result: 84
```

Q2. What is the difference between thenApply(), thenAccept(), and thenRun()?

thenApply(Function) transforms the completed result into a new value, producing a new CompletableFuture<R> that carries the transformed result forward for further chaining.

thenAccept(Consumer) consumes the completed result (e.g., to print it or save it) but does not produce a new usable value - it returns CompletableFuture<Void>.

thenRun(Runnable) runs a follow-up action that doesn't need the result at all - it doesn't even receive the completed value as an argument - also returning CompletableFuture<Void>.

All three also have Async variants (thenApplyAsync, thenAcceptAsync, thenRunAsync) which explicitly run the follow-up action on a separate thread from the common ForkJoinPool (or a supplied Executor), rather than potentially running on the thread that completed the previous stage.

Q3. How do you combine multiple independent CompletableFutures, and what is the difference between thenCombine() and allOf()?

thenCombine(otherFuture, BiFunction) combines exactly two independent CompletableFutures once both complete, merging their two results into a single new value using the provided function - useful when you need to combine two specific, differently-typed results.

CompletableFuture.allOf(future1, future2, ..., futureN) waits for an arbitrary number of futures to all complete, but its return type is CompletableFuture<Void> - it doesn't automatically aggregate their individual results, so each future's own get() must still be called separately afterward to retrieve individual values. There is also anyOf(...), which completes as soon as any one of the given futures completes, useful for "first response wins" scenarios.

CODE EXAMPLE

```
CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 10);
CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(() -> 20);

CompletableFuture<Integer> combined = f1.thenCombine(f2, Integer::sum);
System.out.println(combined.join()); // blocks only here, for demonstration
```

OUTPUT

```
30
```

Q4. How does exception handling work in a CompletableFuture pipeline?

exceptionally(Function) provides a fallback value if any prior stage in the pipeline threw an exception, similar in spirit to a catch block, but the recovery function only runs on the failure path.

handle(BiFunction) runs regardless of whether the previous stage succeeded or failed, receiving both the result (or null if it failed) and the exception (or null if it succeeded), letting you handle both outcomes in one place.

whenComplete(BiConsumer) is similar to handle() (runs on both success and failure, receiving both the result and the exception) but is used purely for side effects like logging, since it does not transform the result and simply passes the original outcome through unchanged.

CODE EXAMPLE

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    if (true) throw new RuntimeException("Something failed");
    return 42;
}).exceptionally(ex -> {
    System.out.println("Recovered from: " + ex.getMessage());
    return -1; // fallback value
});

System.out.println(future.join());
```

OUTPUT

```
Recovered from: java.lang.RuntimeException: Something failed
-1
```

Keep Growing

Thank you for studying with this SkilltoGrowth interview guide. Mastery of Java 8+ features — functional programming with Streams and lambdas, safer null-handling with Optional, the modern Date & Time API, and asynchronous programming with CompletableFuture — signals to interviewers that you write idiomatic, modern Java, not just code that happens to compile. Good luck with your interviews!