# Java 8 Revision Guide

Java 8 remains the dominant Long-Term Support (LTS) release in the Java ecosystem, so employers routinely expect candidates to master both its brand-new features and the traditional core APIs. This revision guide moves from the absolute basics to advanced asynchronous, functional, and parallel programming, giving you a single, citation-rich backbone for last-minute interview study and ongoing skills development.

## JVM Basics and the Java Toolchain

| Term | Description |
|---|---|
| **JVM** | The Java Virtual Machine interprets bytecode and provides services such as garbage collection[^1]. |
| **JDK** | Java Development Kit: JVM + compiler (`javac`), debugger, docs, and tools for developers[^1]. |
| **JRE** | Java Runtime Environment: JVM + core libraries required to run, not compile, applications[^1]. |

### Source → Bytecode Workflow

1. Write `.java` source.
2. `javac` compiles to `.class` bytecode.
3. JVM loads bytecode, just-in-time (JIT) compiles hotspots to native code for the host CPU[^1].

## Core Language Foundations

### Syntax Quick-Hit List

- **Variables & Types** – primitives (`int`, `double`, `boolean`) and reference types (`String`, arrays, objects)[^1].
- **Operators** – arithmetic, logical, bitwise, ternary; beware integer division truncation[^1].
- **Control Flow** – `if/else`, `switch`, enhanced `for`, `while`, `do-while`, labelled `break/continue`[^1].
- **Methods** – pass-by-value (object *references* are copied), varargs, overloading[^1].

### Object-Oriented Pillars

| Pillar | Key Idea | Java Mechanism |
|---|---|---|
| Encapsulation | Hide state behind methods | access modifiers: `private`, getters/setters[^2] |
| Inheritance | Reuse behavior vertically | `extends` for classes, single-inheritance[^3] |
| Polymorphism | Same call, different runtime type | method overriding + dynamic dispatch[^3] |
| Abstraction | Expose *what*, hide *how* | abstract classes & interfaces[^4] |

> **Tip – Interview Cue**: If asked why Java forbids multiple class inheritance, note the *Diamond Problem* and show how Java 8 default methods solve the same issue for interfaces[^5].

## Exception Handling Essentials

1. **Checked vs Unchecked** – checked exceptions extend `Exception` and must be declared or caught; unchecked extend `RuntimeException`[^6].
2. **try-with-resources** – auto-closes any `AutoCloseable`, preventing leaks; always runs *before* `catch` / `finally`[^7].
3. **Custom Exceptions** – extend `Exception` or `RuntimeException`; provide serialVersionUID for best practice[^6].

```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
} catch (IOException ex) {
    throw new DataAccessException(ex);  // custom wrapper
}
```

## Generics Recap

- **Type Safety** – compiler enforces types, eliminating most `ClassCastException` at runtime[^8].
- **Erasure** – generic info is stripped after compilation; use *bounded* wildcards (`<? extends Number>`) to maintain API flexibility[^9].
- **Generic Methods** – declare `<T>` before return type[^8].

```
public static <T extends Comparable<T>> T max(T a, T b){
    return a.compareTo(b) >= 0 ? a : b;
}
```

## Java Collections Framework (JCF)

| Interface | Ordered? | Allows duplicates? | Typical Implementation |
|---|---|---|---|
| `List` | Yes | Yes | `ArrayList`, `LinkedList`[^10] |
| `Set` | No | No | `HashSet`, `TreeSet`[^10] |
| `Map` | Key-value | Keys unique | `HashMap`, `TreeMap`[^10] |
| `Queue` | FIFO/LIFO | Yes | `ArrayDeque`, `PriorityQueue`[^11] |

`Collections.unmodifiableList()` creates shallow, read-only views – common interview favourite[^10].

## Lambda Expressions

```
Comparator<Person> byAge = (p1, p2) -> p1.getAge() - p2.getAge();
```

- **Functional Interface** – exactly one abstract method; use `@FunctionalInterface` for clarity[^12].
- **Syntax** – `(param1, param2) -> expression` or block `{}` with `return`[^13].
- **Variable Capture** – only *effectively final* outer variables may be referenced[^12].

- **Method References** – `String::toUpperCase` (instance), `Integer::parseInt` (static), `Person::new` (constructor)[^13].

## Streams API (Sequential)

```
double avg = people.stream()              // source
                   .filter(p -> p.isAdult()) // intermediate op
                   .mapToInt(Person::getAge)
                   .average()               // terminal op
                   .orElse(0);
```

1. **Pipeline** – lazy intermediate ops build a description; terminal op triggers processing[^14].
2. **Stateless vs. Stateful** – `filter()` is stateless; `sorted()` requires state and may limit parallelism[^15].
3. **Reduction** – `reduce()`, `collect()`, `groupingBy()` for aggregation[^14].

## Parallel Streams

```
long count = Files.lines(bigFile)   // lazy IO
                  .parallel()       // ForkJoin common pool
                  .filter(line -> line.contains("ERROR"))
                  .count();
```

- Backed by the common `ForkJoinPool`; high overhead, so large *CPU-bound* workloads or expensive I/O benefit most[^16].
- Avoid shared mutable state; side-effects break associativity and produce race conditions[^17].
- Check `ForkJoinPool.commonPool().getParallelism()` to understand default thread count[^18].

## Optional

| Pattern | Purpose | Example |
|---|---|---|
| `Optional.of(x)` | Non-null guarantee | Optional.of("ok")[^19] |
| `Optional.empty()` | Explicit absence | Optional.empty()[^19] |
| `map/flatMap` | Compose optionals | userOpt.flatMap(User::getAddress)[^20] |
| `orElse / orElseGet` | Fallback values | cfg.orElse("default")[^20] |

Never call `get()` without `isPresent()` – interviewers love this trap[^20].

## Interface Default & Static Methods

```
interface SmartDevice {
    default void selfCheck(){ System.out.println("All OK"); }
    static int maxId(int a, int b){ return Math.max(a,b); }
}
```

- **Extends Without Breakage** – lets libraries add behaviour without forcing user classes to implement new methods[^21].
- **Diamond Resolution** – implementer must override conflicting defaults or qualify with `InterfaceName.super.method()`[^22].
- Static methods **are not inherited**; call via `InterfaceName.method()`[^23].

## java.time API (JSR-310)

| Class | Purpose | Thread-safe? |
|---|---|---|
| `LocalDate` | Date without time zone | Yes[^24] |
| `LocalTime` | Time without date | Yes[^25] |
| `LocalDateTime` | Date + time | Yes[^24] |
| `ZonedDateTime` | Date-time with zone | Yes[^26] |
| `Period / Duration` | Human / machine time spans | Yes[^24] |

```
LocalDate payday = LocalDate.now().with(TemporalAdjusters.lastDayOfMonth());
```

The package is immutable, ISO-8601 compliant, and replaces the thread-unsafe `java.util.Date`/`Calendar`[^26].

## CompletableFuture & Asynchronous Flows

```
CompletableFuture<IntSummaryStatistics> statsF =
    CompletableFuture.supplyAsync(() -> loadOrders())        // async IO
                     .thenApply(list -> list.stream()
                                            .mapToInt(Order::getTotal)
                                            .summaryStatistics())
                     .exceptionally(ex -> new IntSummaryStatistics());
```

| Method | Role |
|---|---|
| `runAsync` / `supplyAsync` | Fire an async task[^27] |
| `thenApply` | Transform result synchronously[^28] |
| `thenCompose` | Flat-map nested `CompletableFuture`[^29] |
| `thenCombine` | Merge two independent futures[^27] |
| `allOf / anyOf` | Wait for many futures[^30] |
| `exceptionally` / `handle` | Recovery paths[^28] |

`CompletableFuture` extends `Future` **plus** a full `CompletionStage` graph, enabling non-blocking pipelines[^31].

## ExecutorService & Modern Concurrency

```
ExecutorService pool = Executors.newFixedThreadPool(8);
Future<Integer> sumF = pool.submit(() -> compute());
int result = sumF.get(); // blocks
pool.shutdown();
```

- Decouple *task submission* from *thread management*, avoid manual `Thread` creation[^32].
- Prefer `submit(Callable)` + `Future` for a return value; migrate to `CompletableFuture` for fluent async logic[^33].
- Always `shutdown()` gracefully; use `shutdownNow()` only for cancellation emergencies[^34].

## Multithreading Refresher

| Mechanism | Use Case |
|---|---|
| `synchronized` | Mutual exclusion on critical sections[^35] |
| `volatile` | Visibility for single read/write primitives[^36] |
| `java.util.concurrent` | High-level constructs: `Locks`, `Atomic*`, `CountDownLatch`[^33] |
| Thread creation | `extends Thread` *or* `implements Runnable` (prefer)[^37] |

## Other Java 8 Goodies

- **Base64** – `java.util.Base64` for URL-safe or MIME encodes[^38].
- **Arrays.parallelSort()** – parallel quick-sort for primitive arrays[^39].
- **Repeatable Annotations** – annotate same type multiple times; requires a `@Container` annotation[^38].
- **Nashorn** – lightweight JavaScript engine: `jdk.nashorn.api.scripting.NashornScriptEngineFactory`[^38].

## Quick Comparison Tables

### Interface vs Abstract Class

| Feature | Interface (Java 8) | Abstract Class |
|---|---|---|
| Multiple inheritance | Yes[^40] | Single[^41] |
| Method bodies | `default`/`static` only[^21] | Any non-`abstract` method[^22] |
| Constructors | None[^40] | Allowed[^41] |
| Fields | `public static final` only[^40] | Any modifier[^41] |

### Future vs CompletableFuture

| Feature | Future | CompletableFuture |
|---|---|---|
| Check completion | `isDone()` only[^27] | callbacks via `then*` methods[^27] |
| Manual completion | Not possible[^27] | `complete()` / `completeExceptionally()`[^28] |
| Combining tasks | Manual blocking | `thenCompose`, `thenCombine`, `allOf`[^28] |

### Sequential vs Parallel Stream

| Aspect | Sequential | Parallel |
|---|---|---|
| Threading | Single | `ForkJoinPool`[^42] |
| Overhead | Low | High for small workloads[^16] |
| Ordering | Preserved | Not guaranteed unless `forEachOrdered`[^18] |
| Best for | IO or small CPU tasks | CPU-bound, large data, stateless ops[^17] |

## Common Pitfalls & Interview Traps

- Calling `Optional.get()` when empty triggers `NoSuchElementException`[^20].
- Relying on parallel streams with *stateful* lambdas causes data races[^17].
- Forgetting to close an `ExecutorService` leaks threads; always `shutdown()`[^34].
- Using blocking calls (e.g., `Future.get()`) inside a reactive pipeline defeats non-blocking architecture[^27].

## Best-Practice Summary

- Favor immutability and stateless lambdas to maximise safe parallelism[^18].
- Convert *data* to streams at the last possible moment and collect at the last possible moment—maintains pipeline laziness[^14].
- Wrap potentially absent returns in `Optional`, not `null`; never expose raw `Optional` fields[^20].
- Profile before parallelising; thread over-subscription can slow throughput due to context switching[^16].
- Prefer `CompletableFuture` (or higher-level reactive APIs) over manual thread management for IO latency hiding[^28].

## Rapid-Fire Code Cheatsheet

```
// 1. Default + static interface
interface Monitor {
    default boolean isHealthy(){ return true; }
    static long timestamp(){ return System.currentTimeMillis(); }
}

// 2. Filter + map + collect
List<String> ids = logs.stream()
                       .filter(L::isError)
                       .map(L::getId)
                       .distinct()
```

```
                        .collect(Collectors.toList());

// 3. Parallel sum with reduction
long sum = LongStream.rangeClosed(1, 1_000_000)
                     .parallel()
                     .reduce(0, Long::sum);

// 4. Chain async tasks
CompletableFuture<Void> mailF =
    fetchTemplate()
    .thenCompose(tpl -> personalize(tpl, user))
    .thenCompose(text -> sendMailAsync(user, text))
    .exceptionally(ex -> { log.error(ex); return null; });
```

## Final Thoughts

Mastering Java 8 means uniting two mindsets: traditional object-oriented design and modern functional, asynchronous thinking. Review the tables, run the code snippets, and rehearse the pitfalls so you can discuss trade-offs confidently in any technical interview or code review. Good luck upgrading both your skills and your career path!

⁂

[^56]: https://kanyashreecollege.ac.in/pdf/study material/thread.pdf [^57]: https://docs.hazelcast.com/hazelcast/5.5/computing/executor-service [^58]: https://beginnersbook.com/java-tutorial-for-beginners-with-examples/ [^59]: https://www.freecodecamp.org/news/java-collections-framework-reference-guide/ [^60]: https://www.youtube.com/watch?v=K1iu1kXkVoA [^61]: https://www.w3schools.com/java/java_classes.asp [^62]: https://raygun.com/blog/oop-concepts-java/ [^63]: https://www.geeksforgeeks.org/java/polymorphism-in-java/ [^64]: https://www.geeksforgeeks.org/java/implement-interface-using-abstract-class-in-java/ [^65]: https://java-iitd.vlabs.ac.in/exp/exceptions/theory.html [^66]: https://www.datacamp.com/doc/java/classes-and-objects [^67]: https://stackify.com/oops-concepts-in-java/ [^68]: https://www.codecademy.com/learn/learn-java/modules/learn-java-inheritance-and-polymorphism/cheatsheet [^69]: https://www.w3schools.com/java/java_try_catch.asp [^70]: https://www.geeksforgeeks.org/object-class-in-java [^71]: https://www.baeldung.com/java-oop [^72]: https://www.geeksforgeeks.org/cpp/difference-between-inheritance-and-polymorphism/ [^73]: https://www.javacodegeeks.com/2024/08/mastering-completablefuture-in-java-a-comprehensive-guide.html [^74]: https://www.scaler.com/topics/default-method-in-java/ [^75]: https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/default-methods-in-interfaces [^76]: https://concurrencydeepdives.com/guide-completable-future/ [^77]: https://www.youtube.com/watch?v=CB1x1hnh1aE [^78]: https://www.javaguides.net/2024/06/java-completablefuture-tutorial.html [^79]: https://www.codingshuttle.com/blogs/a-comprehensive-guide-to-java-completable-future/ [^80]: https://www.youtube.com/watch?v=xpjvY45Hbyg