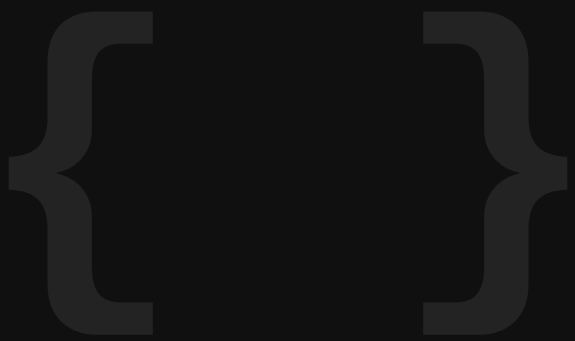
JAVA EXCEPTION HANDLING

Java Exceptions Zero to Hero

A plain-English, example-first guide to understanding, handling, and mastering exceptions in Java

20 PAGES • 16 MODULES • INTERVIEW Q&A • CHEAT SHEET



Contents

Sixteen short modules. Read in order — each one builds on the last.

HOW TO USE THIS GUIDE

Every module follows the same rhythm: a plain-English analogy, the real Java syntax, a short code example, and a key point to remember. Short on time? Re-read just the black boxes on each page — that's the whole module compressed.

WHO THIS IS FOR

Anyone who can already write basic Java (variables, methods, loops) but freezes when a red stack trace shows up. By the last page you'll read any stack trace, decide what to catch, and design your own exception classes with confidence.

01
MODULE

What Is an Exception, Really?

The plumbing that keeps a small problem from becoming a program crash

ANALOGY

Think of your program as a recipe you're following step by step. Halfway through, you reach for an egg and the carton is empty. You don't burn the kitchen down — you pause, deal with the missing egg (swap it, or go buy one), and then continue cooking. An exception is Java's way of saying "something unexpected happened while following the recipe" — it pauses normal execution so you get a chance to react, instead of the whole program crashing silently.

The technical definition

An exception is an object that represents an event which disrupts the normal flow of a program's instructions. When something goes wrong inside a method — dividing by zero, reading a file that isn't there, calling a method on a null reference — Java creates an exception object describing what happened and hands control to code that knows how to deal with it. This process is called throwing an exception, and the code that reacts to it is said to catch it.

What happens if nobody catches it?

If no code catches the exception, it keeps travelling backward through the chain of method calls (this is called propagation) until it reaches `main()`. If it escapes even that, the Java Virtual Machine (JVM) stops the thread and prints a stack trace — the wall of red text every beginner fears. It isn't random noise: it's a breadcrumb trail showing exactly which line failed and which methods called which, read top to bottom, most-recent-call-first.

MINIMAL EXAMPLE

```
public class Demo {
    public static void main(String[] args) {
        int[] marks = {90, 85, 76};
        System.out.println(marks[5]);   // index doesn't exist
        System.out.println("This line never runs");
    }
}

// Output:
// Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
//   Index 5 out of bounds for length 3
//     at Demo.main(Demo.java:4)
```

KEY POINT

An exception is just an object (an instance of a class). That single fact explains almost everything else in this guide: exceptions can be typed, extended, stored in variables, and carry their own data — because they are ordinary Java objects with one special power (they can be "thrown").

02
MODULE

The Exception Family Tree

Every exception you'll ever meet fits into one map

Every throwable object in Java descends from a single class: `Throwable`. It splits into two branches almost immediately — one for problems your code is expected to survive, and one for problems that usually mean the JVM itself is in trouble.

THE HIERARCHY

```
java.lang.Object
   |
   +-- java.lang.Throwable
         |
         +-- java.lang.Error                (JVM / system-level — don't catch)
         |      +-- OutOfMemoryError
         |      +-- StackOverflowError
         |
         +-- java.lang.Exception             (application-level — you handle these)
                +-- IOException                    [CHECKED]
                +-- SQLException                   [CHECKED]
                +-- java.lang.RuntimeException      [UNCHECKED]
                       +-- NullPointerException
                       +-- ArrayIndexOutOfBoundsException
                       +-- ArithmeticException
                       +-- ClassCastException
                       +-- NumberFormatException
```

Reading the tree

- Throwable — the root. Only objects of this type (or a subclass) can be thrown with `throw`.
- Error — signals a serious problem outside your control, like the JVM running out of memory. Your code should generally not try to catch these.
- Exception — problems your application logic can reasonably anticipate and recover from. This is the branch you'll work in 95% of the time.
- RuntimeException — a special sub-branch of Exception that the compiler does not force you to handle. More on this on the next page.

ANALOGY

Picture a company org chart. Throwable is the CEO. Error and Exception are two VPs reporting to the CEO. Everything below them — NullPointerException, IOException, and so on — are just more specific job titles further down the same reporting line. Catching a class also catches everything beneath it in the tree, exactly like a VP's authority covers their whole team.

03 MODULE

Checked vs. Unchecked Exceptions

The single most-asked interview question on this topic

ANALOGY

A checked exception is like airport security before an international flight: the airline (the compiler) refuses to let you board unless you've already planned for it — you must either handle it or declare it. An unchecked exception is like slipping on a wet floor at home: nobody forces you to put up a warning sign in advance; if it happens, it happens, and it's usually a sign you made a mistake (a bug) rather than an unavoidable real-world event.

Aspect	Checked Exception	Unchecked Exception
Parent class	Exception (not RuntimeException)	RuntimeException
Compiler enforcement	Must catch or declare (throws)	No compiler requirement
Represents	Recoverable, foreseeable conditions	Programming mistakes / bugs
Typical cause	External world: files, network, DB	Bad logic: null check missed, bad cast
Examples	IOException, SQLException	NullPointerException, ArithmeticException
When to use for your own class	Caller can realistically recover	Caller can't do much except fix code

Why does this distinction even exist?

Java's designers wanted a way to force developers to think about failure modes that are outside their control — a missing file, a dropped network connection — while not burdening every single line of code with error-handling for mistakes that better tests and better logic should simply prevent. That's the whole philosophy in one sentence: checked = the world might fail you, unchecked = you might have failed yourself.

COMPILER BEHAVIOUR

```
// Checked — this will NOT compile without handling it:
FileReader fr = new FileReader("data.txt");   // throws IOException

// You must do ONE of the following:
try {
    FileReader fr = new FileReader("data.txt");
} catch (IOException e) { /* handle it */ }

// ...or declare it and push the responsibility to the caller:
void readFile() throws IOException { ... }
```

04 MODULE

try / catch / finally — How It Actually Flows

Understand the order of execution, not just the syntax

The `try` block holds code that might fail. The `catch` block holds the recovery plan for a specific exception type. The `finally` block holds cleanup code that must run no matter what — success, failure, or even a `return` statement halfway through.

SYNTAX + EXECUTION ORDER

```
try {
    riskyOperation();       // 1. runs first
} catch (SomeException e) {
    handleError(e);         // 2. runs ONLY if step 1 threw a matching exception
} finally {
    cleanUp();              // 3. ALWAYS runs — after try, or after catch
}
```

Three possible paths through this block

- No exception: try runs fully -> finally runs -> execution continues after the block.
- Exception thrown & caught: try stops at the failing line -> matching catch runs -> finally runs.
- Exception thrown & NOT caught: try stops -> finally still runs -> exception then propagates upward.

KEY POINT

`finally` runs even if the `try` or `catch` block contains a `return` statement. This makes it the correct place for cleanup — closing a file, releasing a database connection, unlocking a resource — because you can guarantee it will execute exactly once.

PROOF: finally RUNS BEFORE THE RETURN VALUE LEAVES

```
static int test() {
    try {
        return 1;
    } finally {
        System.out.println("finally ran");   // prints BEFORE the method returns
    }
}
// Calling test() prints "finally ran" and then returns 1.
```

05
MODULE

Catching More Than One Exception

Order matters, and Java 7 gave us a shortcut

Multiple catch blocks

A single try block can be followed by several catch blocks, each handling a different exception type. Java checks them top to bottom and runs the first one that matches — so more specific exception types must always be listed before their more general parent types.

CORRECT ORDER — SPECIFIC BEFORE GENERAL

```
try {
    process(data);
} catch (NumberFormatException e) {      // specific — checked first
    System.out.println("Bad number format");
} catch (IllegalArgumentException e) {    // NumberFormatException's parent
    System.out.println("Bad argument");
} catch (Exception e) {                   // most general — checked last
    System.out.println("Something else went wrong");
}
```

COMMON MISTAKE

Placing `catch (Exception e)` before a more specific catch block is a compile-time error in Java — the compiler knows the specific block would become unreachable "dead code" and refuses to build. If you ever see this error, reorder your catch blocks from most specific to most general.

Multi-catch (Java 7+): one block, several types

When two or more exception types need the exact same handling logic, combine them with a pipe `|` instead of repeating the block. The caught variable is implicitly final.

MULTI-CATCH SYNTAX

```
try {
    convertAndDivide(input);
} catch (NumberFormatException | ArithmeticException e) {
    System.out.println("Input problem: " + e.getMessage());
}
```

KEY POINT

You cannot multi-catch two exception types where one is a subclass of the other (e.g. `IOException | FileNotFoundException` is a compile error, since FileNotFoundException already IS-A IOException). Java flags this as redundant.

06 MODULE

try-with-resources (Auto-Closing)

Never forget to close a file, socket, or connection again

ANALOGY

Before Java 7, closing a resource was like borrowing library books and having to personally remember to return every single one, even if you tripped and dropped your bag on the way out (an exception). try-with-resources is a librarian who automatically checks every book back in for you the moment you leave the building — success or trip, it doesn't matter.

The old, error-prone way

BEFORE JAVA 7 — verbose and easy to get wrong

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("data.txt"));
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (br != null) {
        try { br.close(); } catch (IOException e) { /* ugh */ }
    }
}
```

The modern way

JAVA 7+ — resource closes automatically

```
try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
// br.close() is called automatically here — even if an exception was thrown.
```

KEY POINT

Any object you place inside the parentheses must implement the `AutoCloseable` interface (which defines a single method, `close()`). Most I/O classes — files, streams, database connections — already implement it. You can make your own classes work this way too by implementing it yourself.

07
MODULE

throw vs. throws — The Two Cousins

One word, two completely different jobs

These two keywords look almost identical and confuse nearly every beginner at least once. The trick is to remember that one is a verb performed inside a method body, and the other is a label on the method signature.

	throw	throws
Used where	Inside a method or block	In a method's signature (header)
Purpose	Actually raises one exception, right now	Declares which exception types the method might raise
Followed by	A single exception instance	One or more exception class names, comma-separated
Count	Exactly one per statement	As many as needed
Example	throw new IOException("...");	void save() throws IOException

BOTH KEYWORDS WORKING TOGETHER

```
// "throws" warns callers this method might raise IOException
void saveToDisk(String path) throws IOException {
    if (path == null || path.isEmpty()) {
        // "throw" actually raises the exception, right here, right now
        throw new IOException("Path cannot be empty");
    }
    // ... write logic ...
}
```

ANALOGY

`throws` is a warning label on a package — "Fragile: may contain broken glass." It tells the person handling it (the caller) what to be ready for. `throw` is the moment the glass actually breaks.

08 MODULE

Building Your Own Exception

Give your errors a name that matches your business logic

Built-in exceptions are generic on purpose. Real applications benefit from exceptions with names that describe your domain — `InsufficientFundsException` communicates far more than a generic `IllegalStateException` ever could, both to future developers and to anyone reading a log file at 2 AM.

Step 1 — pick a parent class

- Extend Exception if callers should be forced to handle it (checked).
- Extend RuntimeException if it represents a programming error that shouldn't need explicit handling everywhere (unchecked).

A CHECKED CUSTOM EXCEPTION

```
public class InsufficientFundsException extends Exception {
    private final double shortfall;

    public InsufficientFundsException(String message, double shortfall) {
        super(message);
        this.shortfall = shortfall;
    }

    public double getShortfall() {
        return shortfall;
    }
}
```

USING IT

```
void withdraw(double amount) throws InsufficientFundsException {
    if (amount > balance) {
        throw new InsufficientFundsException(
            "Not enough balance", amount - balance);
    }
    balance -= amount;
}
```

KEY POINT

Always provide a constructor that accepts a message (and ideally a cause — see the next module). Adding extra fields, like `shortfall` above, is what makes a custom exception genuinely more useful than reusing a generic one — the catch block gets real, structured data to act on, not just a string.

09
MODULE

Exception Chaining

Never let the original cause of a bug get lost in translation

ANALOGY

Imagine a doctor's referral letter. A general physician notices something and refers you to a specialist, but the letter still includes the original notes — it doesn't say "patient has a problem" and throw away the details. Exception chaining does the same thing: when you catch a low-level exception and throw a higher-level one to describe it in your application's terms, you attach the original as the cause so nobody loses the trail.

WITHOUT CHAINING — the original cause is lost

```
try {
    readConfigFile();
} catch (IOException e) {
    throw new RuntimeException("Startup failed");   // original 'e' is gone!
}
```

WITH CHAINING — the original cause is preserved

```
try {
    readConfigFile();
} catch (IOException e) {
    throw new RuntimeException("Startup failed", e);   // 'e' passed as the cause
}

// Later, anyone catching this can still inspect the root cause:
catch (RuntimeException e) {
    Throwable rootCause = e.getCause();     // returns the original IOException
}
```


Why this matters in practice

A stack trace with a preserved cause shows both exceptions, joined by the line "Caused by:" — giving you the full story: what your application-level code was trying to do, and the low-level technical reason it failed. This single habit saves enormous debugging time in any real, multi-layered application.

KEY POINT

Almost every built-in exception constructor has an overload that accepts a `Throwable cause` as the last argument. Get in the habit of using it every time you catch-and-rethrow.

10 MODULE

The 10 Exceptions You'll Meet Every Day

Recognise them on sight — no more panic when you see the name

Exception	Fires when...	Typical fix
NullPointerException	You call a method or access a field on a reference that is null	Null-check before use, or use Optional / Objects.requireNonNull
ArrayIndexOutOfBoundsException	You access an array index that doesn't exist (< 0 or >= length)	Validate the index against array.length before use
StringIndexOutOfBoundsException	Same idea, but on a String (charAt, substring, etc.)	Check index against string.length()
ClassCastException	You force-cast an object to a type it isn't actually an instance of	Use 'instanceof' to check before casting
NumberFormatException	You parse text that isn't a valid number, e.g. Integer.parseInt("abc")	Validate the string format first, or catch and handle
ArithmeticException	Integer division by zero (e.g. 10 / 0)	Check the divisor is non-zero before dividing
IllegalArgumentException	A method receives an argument that is invalid for its logic	Validate arguments at the start of the method
IllegalStateException	A method is called at a time when the object isn't ready for it	Check object state before calling the method
ConcurrentModificationException	You modify a collection while iterating over it directly	Use an Iterator's remove(), or a CopyOnWriteArrayList
UnsupportedOperationException	You call a method an implementation deliberately doesn't support	Check the docs — some collections are read-only

PRO TIP

Notice that the fix column is almost always some form of "check before you act." The vast majority of unchecked exceptions in real code are prevented not by clever catch blocks, but by a simple validation one line earlier.

11 MODULE

Errors You Can't (Usually) Catch

When the problem is bigger than your code

Recall from the family tree on page 4: `Error` is a separate branch from `Exception`. Errors represent conditions serious enough that recovery inside normal application code usually isn't realistic — the JVM itself is running out of a critical resource.

Error	What it means	What to actually do
StackOverflowError	A method called itself (directly or indirectly) too many times — usually infinite/runaway recursion	Fix the recursion: add or correct the base case
OutOfMemoryError	The JVM heap is full and garbage collection can't free enough space	Fix memory leaks; increase heap size only as a last resort
NoClassDefFoundError	A class was present at compile time but missing at runtime	Check your classpath / dependency configuration
ExceptionInInitializerError	A static initializer block or static field initializer threw	Debug the static initialization logic

A CLASSIC StackOverflowError

```
static int factorial(int n) {
    return n * factorial(n - 1);   // missing base case — never stops!
}

factorial(5);
// Exception in thread "main" java.lang.StackOverflowError
//     at Demo.factorial(Demo.java:2)
//     at Demo.factorial(Demo.java:2)   <- repeats thousands of times
```

KEY POINT

Technically you can write `catch (Error e)` — the compiler allows it, because Error extends Throwable just like Exception does. But by convention you almost never should: catching an Error usually just delays an inevitable crash while hiding the real, underlying problem from you.

12 MODULE

Tricky Gotchas That Trip Up Beginners

The questions that separate a beginner from someone who really gets it

Gotcha 1 — finally can silently override a return

A TRAP

```
static int trap() {
    try {
        return 1;
    } finally {
        return 2;      // this OVERRIDES the try block's return value!
    }
}
// trap() returns 2, not 1. Avoid 'return' inside finally.
```

Gotcha 2 — swallowing an exception

DON'T DO THIS

```
try {
    riskyCall();
} catch (Exception e) {
    // empty block — the exception vanishes with zero trace
}
```

An empty catch block is one of the most common causes of "impossible" bugs in production — the program keeps running in a broken state with no clue why. At minimum, log the exception.

Gotcha 3 — catching Exception instead of a specific type

Catching the broad `Exception` class hides bugs you didn't anticipate (including RuntimeExceptions that reveal real logic errors) behind the same generic handling meant for expected, recoverable problems. Catch the most specific type you can.

Gotcha 4 — checking with == instead of catching

Beginners sometimes try to prevent a NullPointerException with `if (obj != null)` everywhere, which is good — but then still leave a NumberFormatException unguarded a few lines later on user input. Defensive checks and exception handling are complementary, not either/or; use whichever is more direct for each specific risk.

13 MODULE

Best Practices — Do's and Don'ts

How experienced Java developers actually write this code

DO

- Catch the most specific exception type your logic can meaningfully react to.
- Always include a clear, actionable message when you throw — future you will read it at 2 AM.
- Use try-with-resources for anything that implements AutoCloseable.
- Chain the original cause whenever you wrap one exception in another.
- Validate inputs early and throw IllegalArgumentException with a message naming the bad value.
- Log exceptions at the point where you have enough context to act — not several layers away.

DON'T

- Don't leave a catch block empty — it hides real bugs from everyone, including future you.
- Don't use exceptions for normal control flow (e.g. throwing to "break" out of a loop early).
- Don't catch `Throwable` or `Error` just to keep the program alive at any cost.
- Don't create a custom exception for every tiny variation — group related failures sensibly.
- Don't print a stack trace and rethrow and log it again three layers up — pick one place.

PRO TIP

A good litmus test before writing a try/catch: "If this exception happens, what will my code actually DO differently?" If the honest answer is "nothing, I just want the red error to go away," that's a sign you're about to hide a bug instead of handling it.

14 MODULE

Exceptions in Real Projects

How this looks once you leave toy examples behind

Layered exception handling

In a typical multi-layer application (say, a REST API backed by a database), each layer usually translates low-level exceptions into ones meaningful at its own level, chaining the cause as it goes:

A REALISTIC THREE-LAYER FLOW

```
// Data layer: talks to the database, throws low-level exceptions
public User findUser(int id) throws SQLException { ... }

// Service layer: translates into a domain-specific exception
public User getUser(int id) {
    try {
        return repository.findUser(id);
    } catch (SQLException e) {
        throw new UserLookupException("Could not load user " + id, e);
    }
}

// Web layer: converts the domain exception into an HTTP response
catch (UserLookupException e) {
    return ResponseEntity.status(404).body("User not found: " + e.getMessage());
}
```

A note on logging

Use a proper logging framework (Logback, Log4j, java.util.logging) instead of `System.out.println` or `printStackTrace()` in real projects. Loggers let you attach severity levels, timestamps, and route output to files or monitoring systems — none of which a print statement can do.

KEY POINT

Each layer should only catch what it can meaningfully translate or resolve, then let everything else propagate. Trying to handle every possible exception at every single layer leads to duplicate, noisy error handling that's harder to maintain than no handling at all.

15 MODULE

Interview Questions — Part 1

Simple words, straight to the point — exactly how to answer out loud

Q1. What is the difference between an Exception and an Error?

An Exception is a problem your application can usually anticipate and recover from (bad input, missing file). An Error signals a serious, usually unrecoverable JVM-level problem (out of memory). You handle Exceptions; you generally don't try to catch Errors.

Q2. What is the difference between checked and unchecked exceptions?

Checked exceptions are verified by the compiler — you must either catch them or declare them with 'throws'. Unchecked exceptions (RuntimeException and its subclasses) are not checked by the compiler; they usually represent programming mistakes.

Q3. Can we have a try block without a catch block?

Yes — as long as it's followed by a finally block, or it's a try-with-resources block. A try must be paired with at least one of catch or finally.

Q4. Does finally always execute?

Almost always — including when try or catch contains a return, break, or continue. The only exceptions are if the JVM exits (System.exit()) or the thread is killed while finally is pending.

Q5. What is the difference between throw and throws?

'throw' actually raises one specific exception instance, inside a method body. 'throws' is a keyword in a method's signature declaring which exception types that method might raise.

Q6. Why is catching Exception generally discouraged?

It catches every subclass — including bugs represented by RuntimeExceptions you didn't anticipate — under the same generic handling meant for truly expected, recoverable conditions, making bugs harder to notice and debug.

15 MODULE

Interview Questions — Part 2

Continued — the questions that go one level deeper

Q7. What happens if an exception is thrown inside a catch block?

It propagates upward just like any other exception, unless there's a finally block below it — which still runs first before the new exception continues propagating.

Q8. Can a finally block override an exception?

Yes. If finally itself throws an exception (or contains a return), it replaces/suppresses whatever exception or return value came from the try or catch block. This is why finally should only contain simple, reliable cleanup code.

Q9. What is exception chaining and why does it matter?

It's passing an original exception as the 'cause' of a new one you throw, using a constructor like 'new RuntimeException(message, cause)'. It matters because it preserves the full root cause in the stack trace instead of losing it when you translate exceptions between layers.

Q10. Should custom exceptions extend Exception or RuntimeException?

Extend Exception (checked) when the caller can realistically do something about the failure and should be forced to consider it. Extend RuntimeException (unchecked) when it represents a programming error the caller isn't expected to recover from at that call site.

Q11. What is try-with-resources and what interface does it rely on?

A try variant, introduced in Java 7, that automatically closes any resource declared in its parentheses once the block ends — success or failure. The resource must implement the AutoCloseable interface.

Q12. Is it legal to catch Throwable?

Legally yes, the compiler allows it — but it's almost always bad practice, because it also catches Errors like OutOfMemoryError, which your code usually cannot meaningfully recover from anyway.

16 MODULE

One-Page Cheat Sheet

Everything on this page, memorised, is enough to pass most interviews

The hierarchy, compressed

```
Throwable -> Error (don't catch) | Exception -> Checked (must handle)
                                              -> RuntimeException (unchecked)
```

The five keywords

Keyword	One-line meaning
try	Marks code that might fail
catch	Recovers from one specific exception type
finally	Cleanup that always runs
throw	Actually raises one exception, right now
throws	Declares what a method might raise (in its signature)

Decision shortcuts

- Recoverable, external cause (file, network, DB) -> checked exception.
- Programming mistake (bad null, bad cast, bad math) -> unchecked exception.
- Resource needs closing (file, socket, connection) -> try-with-resources.
- Rethrowing after catching -> always chain the original cause.
- Multiple catch blocks -> order from most specific to most general.
- Deciding a custom exception's parent -> "Can the caller realistically recover?" Yes = Exception. No = RuntimeException.

THE ONE RULE THAT MATTERS MOST

Never let an exception disappear silently. Whether you fix it, log it, wrap it, or rethrow it — always do something visible with it. An empty catch block is the single most expensive habit to unlearn.

MODULE

Practice Challenges & Recap

Zero to Hero — the final checkpoint

Try these five, without looking back

- 1. Write a method that divides two integers and throws a custom `DivisionByZeroException` (checked) instead of the built-in ArithmeticException.
- 2. Write a try-with-resources block that opens two files at once and prints the first line of each.
- 3. Write a chain of three catch blocks for NumberFormatException, IllegalArgumentException, and Exception — in the correct, compilable order.
- 4. Write a method where a finally block logs "cleanup done" and prove, with output, that it runs even when the try block throws.
- 5. Explain out loud, in under 30 seconds, the difference between throw and throws, and checked vs unchecked — using your own analogy, not the one from this guide.

What "Hero" looks like

You started this guide possibly afraid of a red stack trace. By now you should be able to: read any stack trace top to bottom and identify the failing line; decide in seconds whether an exception should be checked or unchecked; write a try-with-resources block from memory; design a custom exception with the right parent class; and — most importantly — know that an empty catch block is never the answer.

KEEP GOING

Exceptions are one piece of writing reliable Java. The natural next topics to study are: the Optional class (reducing null-related exceptions at the source), the Java Collections Framework, and multithreading — where exception handling gets one extra layer of complexity worth mastering next.

End of notes — Java Exceptions: Zero to Hero