Exception Handling Best Practices

Practices built on top of the previous six lessons' mechanics -- the anti-patterns of using exceptions for control flow and swallowing them, catching in the right order from specific to broad, and only catching where a genuinely useful response is possible. The 7th and final lesson in the Exception Handling series, closing with a recap of the whole series.

Beginner 20 min

Last updated: November 10, 2026
Read online: https://www.learnforgex.com/en/topics/exception-handling-best-practices

---

Six lessons in, you know the mechanics of Java exceptions cold: how they're structured, how to catch and finalize, how the hierarchy works, checked vs unchecked, `throw` / `throws`, and designing your own types. Knowing the mechanics is not the same as using them well — this closing lesson is about the habits and anti-patterns that separate code that merely compiles from code a team can actually trust in production.

Why This Lesson Exists at All

Every mechanic covered so far is legal Java — the compiler accepts all of it. But legal isn't the same as wise: an empty `catch` block compiles fine and quietly destroys your ability to debug production failures; catching `Exception` compiles fine and silently swallows bugs alongside real failures. This lesson collects the practices experienced Java developers apply on top of the rules you already know.

Anti-Pattern: Using Exceptions for Control Flow

Exceptions exist for EXCEPTIONAL conditions — outcomes that are not part of a method's normal, expected operation. Using `throw` / `catch` to implement ordinary control flow (like breaking out of a loop once you've found something) abuses a mechanism built for error propagation to do a job `return` or `break` already does, at real runtime cost (building a stack trace is not free) and at a real readability cost.

```
public class ExceptionsForControlFlowAntiPatternExample {

    public static void main(String[] args) {
        int[] numbers = {4, 8, 15, 16, 23, 42};

        System.out.println("bad (exception as control flow): " + findFirstOver20_bad
(numbers));
        System.out.println("good (ordinary control flow):    " + findFirstOver20_good
(numbers));
    }

    // BAD: throwing and catching an exception purely to break out of a loop
    // once a value is found. Nothing here is actually exceptional -- "found
    // a matching number" is a completely normal, expected outcome. Building
    // and unwinding a stack trace for it is wasted work and hides the real
    // control flow behind a try/catch that has nothing to do with error
    // handling.
    static int findFirstOver20_bad(int[] numbers) {
        try {
            for (int n : numbers) {
                if (n > 20) {
                    throw new FoundException(n);
                }
            }
        } catch (FoundException e) {
            return e.value;
        }
        return -1;
    }

    static class FoundException extends RuntimeException {
        final int value;
        FoundException(int value) {
            this.value = value;
        }
    }

    // GOOD: the same result using ordinary control flow -- no exception
    // involved at all, because nothing exceptional happened.
    static int findFirstOver20_good(int[] numbers) {
        for (int n : numbers) {
            if (n > 20) {
                return n;
            }
        }
        return -1;
    }
}
```

Expected Output

```
bad (exception as control flow): 23
good (ordinary control flow):    23
```

`findFirstOver20_bad(...)` throws and catches a `FoundException` purely to exit a loop — finding a matching number is a completely ordinary outcome, not an exceptional one.

`findFirstOver20_good(...)` produces the identical result with a plain `return`, no exception machinery involved at all.

> If you ever find yourself writing a custom exception whose only purpose is to carry a value out of a loop or a deeply nested call, that's a strong signal you're using exceptions for control flow — restructure the code with ordinary `return` values instead.

Anti-Pattern: Swallowing Exceptions

An empty `catch` block is one of the most damaging things you can write in Java: the failure happened, but every trace of it — the type, the message, the stack trace — disappears the instant execution reaches that empty block. Whoever debugs the resulting symptom later has nothing to go on.

```
public class SwallowingExceptionsAntiPatternExample {

    public static void main(String[] args) {
        System.out.println("bad result:  " + parsePort_bad("not-a-number"));
        try {
            parsePort_good("not-a-number");
        } catch (IllegalArgumentException e) {
            System.out.println("good result: rejected with a clear reason -- " + e.
getMessage());
        }
    }

    // BAD: an empty catch block swallows the exception entirely -- the
    // failure leaves no trace anywhere, and the caller gets a silently
    // wrong value (0) with no way to tell it apart from a genuinely
    // configured port 0.
    static int parsePort_bad(String text) {
        int port = 0;
        try {
            port = Integer.parseInt(text);
        } catch (NumberFormatException e) {
            // swallowed -- nothing here at all
        }
        return port;
    }

    // GOOD: if there's truly nothing useful this method can do about the
    // failure, don't catch it at all here -- let it propagate, or catch it
    // only to translate it into a clearer, more specific failure for this
    // method's own caller.
    static int parsePort_good(String text) {
        try {
            return Integer.parseInt(text);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("'" + text + "' is not a valid port number",
e);
        }
    }
}
```

`parsePort_bad(...)` swallows `NumberFormatException` and returns `0` — a caller has no way to tell a genuinely configured port `0` apart from a parse failure that got silently discarded. `parsePort_good(...)` does one of the two honest things: if nothing useful can be done here, don't catch at all (let it propagate); if you DO catch, translate the failure into something clearer for your own caller — a technique you already know as wrapping, from "Throw and Throws."

> If you're tempted to write an empty `catch` block "just to make the code compile" while you figure out the right handling later, at minimum print or log something inside it as a placeholder — an empty block is easy to forget about forever.

Catching Specifically, and in the Right Order

A single `try` can have multiple `catch` blocks, and — as you saw in "Try-Catch and Finally" — Java requires them ordered from most specific to least specific. That compiler rule exists to serve a real design goal: each `catch` block should react to exactly the failure it names, not to a broad category that happens to include it.

```
public class CatchOrderAndSpecificityExample {

    public static void main(String[] args) {
        process("42");
        process("not-a-number");
        process(null);
    }

    static void process(String text) {
        try {
            System.out.println("length: " + text.length() + ", parsed: " + Integer.parseInt
(text));
        } catch (NumberFormatException e) {
            // Specific: this failure has one clear, targeted response.
            System.out.println("'" + text + "' is not a number");
        } catch (NullPointerException e) {
            // Also specific: a completely different failure, a different
            // targeted response -- listing it separately (instead of
            // folding it into a broad catch below) keeps each response
            // honest about what it's actually reacting to.
            System.out.println("no input was provided");
        } catch (RuntimeException e) {
            // A broad catch-all, placed LAST (the compiler would reject
            // it above the more specific catches, since it's a
            // supertype of both). Reserved for genuinely unanticipated
            // failures -- not a substitute for handling the two known
            // ones above.
            System.out.println("unexpected failure: " + e);
        }
    }
}
```

> **Expected Output**
>
> ```
> length: 2, parsed: 42
> 'not-a-number' is not a number
> no input was provided
> ```

`process(...)` catches `NumberFormatException` and `NullPointerException` separately, each with its own targeted response, and only falls back to a broad `catch (RuntimeException e)` for genuinely unanticipated failures — placed last, since the compiler rejects a supertype catch positioned before its subtypes. Catching `Exception` (or worse, `Throwable`) as a matter of habit, instead of as a deliberate last resort, tends to lump together failures that need completely different responses.

Only Catch What You Can Actually Handle

Not every method that CAN catch an exception SHOULD. A `catch` block only earns its place when the method has enough context to do something meaningful with the failure — retry, fall back to a default, translate it for its own caller. If a method has no such response, catching there just to immediately do nothing useful (or to rethrow unchanged) adds code without adding value.

```
public class OnlyCatchWhatYouCanHandleExample {

    public static void main(String[] args) {
        // The top of the call chain is where a retry decision actually
        // makes sense -- it's the only place with enough context (should
        // we ask the user again? give up? fall back to a default?) to
        // decide what "handling" the failure even means.
        try {
            System.out.println("price: " + readPriceWithRetry());
        } catch (IllegalStateException e) {
            System.out.println("giving up: " + e.getMessage());
        }
    }

    static double readPriceWithRetry() {
        int attempts = 0;
        while (attempts < 3) {
            attempts++;
            try {
                return parsePrice(attempts < 3 ? "bad-input" : "19.99");
            } catch (NumberFormatException e) {
                // This layer CAN meaningfully handle the failure -- it
                // knows how to retry, so it catches, reacts, and moves on
                // instead of letting the exception propagate needlessly.
                System.out.println("attempt " + attempts + " failed, retrying...");
            }
        }
        throw new IllegalStateException("could not read a valid price after 3 attempts");
    }

    static double parsePrice(String text) {
        // This method has no useful response to a malformed price -- it
        // isn't the one deciding whether to retry or give up, so it does
        // NOT catch NumberFormatException here. Catching it just to
        // immediately rethrow the same thing would add a try/catch that
        // does nothing.
        return Double.parseDouble(text);
    }
}
```

`parsePrice(...)` deliberately does NOT catch `NumberFormatException` — it has no basis for deciding whether to retry or give up, so it lets the exception propagate. `readPriceWithRetry(...)` is where the catch actually belongs, because it's the layer that knows how to react: retry a few times, then give up with a clear `IllegalStateException` for whoever called it.

Putting It All Together: A Recap of the Series

Across this series you built up the full picture one layer at a time: "Introduction to Exceptions" covered what an exception even is and what happens when one goes uncaught; "Try-Catch and Finally" covered the mechanics of handling and cleanup; "Exception Hierarchy" covered how `Throwable`, `Error`, `Exception`, and `RuntimeException` relate and how `catch` matches by supertype; "Checked vs Unchecked Exceptions" covered the compiler-enforced contract and when to choose which; "Throw and Throws" covered creating a failure at runtime versus declaring one at compile time; "Creating and Throwing Custom Exceptions" covered designing your own types. This lesson's four practices are what turn that full toolbox into code a team can trust: don't reach for `throw`/`catch` when ordinary control flow will do, never let a `catch` block erase the evidence of what went wrong, catch specific types in the right order, and only catch where you can actually respond.

Best Practices

- Reserve exceptions for genuinely exceptional conditions — never for expected, everyday outcomes your code can check for directly.
- Never leave a `catch` block truly empty; at an absolute minimum, note that the failure happened.
- List `catch` blocks from most specific to least specific, and treat a broad `catch (Exception e)` as a deliberate last resort, not a default habit.
- Catch only where a method has a genuine, useful response to the failure — otherwise let it propagate to a layer that does.
- When you do catch and rethrow a different type, always preserve the original as the `cause`, exactly as covered in "Throw and Throws."

Common Mistakes

- Designing a whole custom exception (like a "found" signal) just to break out of a loop or nested call.
- Writing `catch (Exception e) {}` "temporarily" and never coming back to fix it.
- Catching a broad supertype before a narrower one is even considered, collapsing genuinely different failures into one generic response.
- Catching an exception purely to log it and rethrow the exact same thing unchanged — this adds a `try`/`catch` that does nothing a caller couldn't already do itself.

Summary, Cheat Sheet, and Glossary

Summary

- Exceptions are for exceptional conditions, not for implementing ordinary control flow.

- An empty `catch` block destroys the evidence of a failure — never leave one silently empty.
- `catch` blocks should be specific and ordered from most to least specific; a broad catch is a deliberate last resort.
- Only catch where a method can genuinely act on the failure — otherwise, let it propagate.
- The full series builds one coherent toolbox: what exceptions are, handling and cleanup, the hierarchy, checked vs unchecked, throw vs throws, and designing your own types.

Cheat Sheet

```
// Don't: exceptions as control flow
try {
    for (int n : numbers) {
        if (n > 20) throw new FoundException(n);
    }
} catch (FoundException e) { return e.value; }

// Do: ordinary control flow
for (int n : numbers) {
    if (n > 20) return n;
}

// Don't: swallowing
try {
    return Integer.parseInt(text);
} catch (NumberFormatException e) { /* nothing */ }

// Do: translate, don't erase
try {
    return Integer.parseInt(text);
} catch (NumberFormatException e) {
    throw new IllegalArgumentException("invalid: " + text, e);
}

// Do: specific before broad
catch (NumberFormatException e) { ... }
catch (NullPointerException e) { ... }
catch (RuntimeException e) { ... } // last resort
```

Glossary

- **Exceptions as control flow**: an anti-pattern that uses `throw`/`catch` to implement ordinary, expected logic instead of error propagation.
- **Swallowing an exception**: catching it and discarding all trace of it, usually via an empty `catch` block.
- **Catch specificity**: ordering `catch` blocks from the narrowest, most targeted type to the broadest.
- **Handleable**: a failure a given method has enough context to actually react to (retry, fall back, translate) rather than just pass through.