Try-Catch and Finally

The actual mechanism for handling an exception: the basic try-catch block, multiple catch blocks (matching in order, the superclass/subclass ordering restriction), multi-catch with | (Java 7+), finally running unconditionally every time, and how a return /throw inside finally silently overrides whatever try/catch was about to produce. The 2nd lesson in the Exception Handling series.

Beginner 20 min

Generated on 2 October 2026
Read online: https://www.learnforgex.com/en/topics/try-catch-finally

---

"Introduction to Exceptions" showed what happens when nothing handles an exception -- the program terminates, loudly, with a stack trace. This lesson covers the actual mechanism for stepping in before that happens: `try`, `catch`, and `finally`.

What Are try and catch?

A `try` block marks a section of code as "this might throw an exception -- watch it." A `catch` block, immediately following it, specifies what to do if a SPECIFIC type of exception actually occurs inside that `try` block. If no exception is thrown, every `catch` block is simply skipped, and execution continues right after them as if they weren't there.

Why Does It Exist?

"Introduction to Exceptions" already covered why exceptions themselves exist as a language feature. `try` / `catch` answers a narrower, more practical question: once you know a specific piece of code MIGHT fail, how do you contain that failure to exactly the place you're prepared to deal with it, instead of letting it terminate the whole program? Wrapping the risky code in `try`, and the recovery logic in `catch`, keeps the "what if this fails" handling physically next to the code that might actually fail.

History

`try` / `catch` / `finally` as keywords predate Java -- Java's designers borrowed this exact three-part shape directly from C++'s exception handling syntax (added to C++ in the late 1980s), rather than inventing a new one. `finally` itself, guaranteeing cleanup code runs regardless of outcome, is Java's own addition to that borrowed shape -- C++ has no direct equivalent, relying instead on a different pattern (RAII) to guarantee cleanup.

The Basic try-catch Block

```
// Compare this to UncaughtExceptionExample from "Introduction to Exceptions" --
// same idea (dividing by zero), but this time wrapped in a try-catch block.
// The exception is still thrown, still created with the same class and
// message -- the only difference is that something now intercepts it BEFORE
// it reaches main() uncaught, so the program keeps running instead of
// terminating.
public class BasicTryCatchExample {
    public static void main(String[] args) {
        System.out.println("About to divide...");
        int result = safeDivide(10, 0);
        // Unlike UncaughtExceptionExample, this line DOES run -- the
        // exception was handled, so execution continues normally after the
        // try-catch block.
        System.out.println("Result: " + result);
        System.out.println("Program continues normally.");
    }

    private static int safeDivide(int a, int b) {
        try {
            return a / b;   // throws ArithmeticException when b == 0
        } catch (ArithmeticException e) {
            // "e" is the SAME kind of exception object described in
            // "Introduction to Exceptions" -- it has a message, a class, and
            // a stack trace. Here we just read its message.
            System.out.println("Caught it: " + e.getMessage());
            return 0;   // a sensible fallback value
        }
    }
}
```

> The parameter in a `catch` block (`e` above) is a real, ordinary local variable, scoped to that `catch` block only -- you can call any method `Throwable` defines on it (see the "Introduction to Exceptions" lesson's "The Anatomy of an Exception: Throwable, Message, and Stack Trace" section), most commonly `getMessage()`.

Multiple catch Blocks: Matching in Order

A single `try` can be followed by several `catch` blocks, each handling a DIFFERENT exception type -- Java checks them top to bottom and runs only the FIRST one that matches.

```
// A single try block can be followed by SEVERAL catch blocks -- Java checks
// them TOP TO BOTTOM and runs the FIRST one whose type matches the thrown
// exception, then skips every catch block after it. Only ONE catch block
// ever runs per exception, never more than one.
public class MultipleCatchBlocksExample {
    public static void main(String[] args) {
        parseAndDivide("100", "0");     // triggers the ArithmeticException catch
        parseAndDivide("abc", "5");     // triggers the NumberFormatException catch
        parseAndDivide("100", "5");     // triggers neither -- prints the result
    }

    private static void parseAndDivide(String numeratorText, String denominatorText) {
        try {
            int numerator = Integer.parseInt(numeratorText);
            int denominator = Integer.parseInt(denominatorText);
            System.out.println(numeratorText + " / " + denominatorText + " = " + (numerator
/ denominator));
        } catch (ArithmeticException e) {
            // Matches ONLY division-related failures (denominator == 0).
            System.out.println("Cannot divide: " + e.getMessage());
        } catch (NumberFormatException e) {
            // Matches ONLY invalid number text -- NOTE: this catch block
            // must come AFTER any catch for a SUPERCLASS of
            // NumberFormatException, or the compiler rejects it as
            // unreachable (see "Common Mistakes").
            System.out.println("Not a valid number: " + e.getMessage());
        }
    }
}
```

> Order matters. If a `catch` block for a SUPERCLASS (like `RuntimeException`) came before a `catch` block for one of its subclasses (like `NumberFormatException`), the subclass's `catch` block would be unreachable -- the superclass one would always match first. The compiler catches this specific mistake and refuses to build (see "Exception Hierarchy" for what "superclass" means precisely here).

Multi-Catch: Catching Several Exception Types in One Block with |

When two or more DIFFERENT exception types genuinely need the SAME handling code, writing out separate, duplicate `catch` blocks (as in `MultipleCatchBlocksExample`) repeats yourself for no reason. Multi-catch, added in Java 7, lets one `catch` block list several types separated by `|`.

```
// Multi-catch (Java 7+): when two or more DIFFERENT exception types need the
// EXACT SAME handling code, "|" lets one catch block handle all of them,
// instead of duplicating the same body in separate catch blocks (compare to
// MultipleCatchBlocksExample, where the two blocks needed genuinely
// DIFFERENT handling).
public class MultiCatchExample {
    public static void main(String[] args) {
        parseAndDivide("100", "0");     // ArithmeticException -- caught below
        parseAndDivide("abc", "5");     // NumberFormatException -- caught below, SAME
handling
        parseAndDivide("100", "5");     // neither -- prints the result
    }

    private static void parseAndDivide(String numeratorText, String denominatorText) {
        try {
            int numerator = Integer.parseInt(numeratorText);
            int denominator = Integer.parseInt(denominatorText);
            System.out.println(numeratorText + " / " + denominatorText + " = " + (numerator
/ denominator));
        } catch (ArithmeticException | NumberFormatException e) {
            // Both exception types land here -- "e" is typed as their
            // nearest common supertype for anything they share (in this
            // case, RuntimeException -- see "Exception Hierarchy" for what
            // that means precisely). The two types in a multi-catch can
            // NEVER be one a subclass of the other -- the compiler rejects
            // that as redundant.
            System.out.println("Could not complete the calculation: " + e.getMessage());
        }
    }
}
```

finally: The Block That Always Runs

A `finally` block, placed after the last `catch` (or directly after `try`, with no `catch` at all), runs in every case -- whether the `try` block succeeded, an exception was caught, or an exception propagated past every `catch` block uncaught.

```
// finally runs in EVERY case -- whether the try block succeeds, throws an
// exception that gets caught, or (as the last call below shows) throws an
// exception that DOESN'T get caught at all. This is what makes finally the
// right place for cleanup code that must run no matter what happened.
public class FinallyAlwaysRunsExample {
    public static void main(String[] args) {
        System.out.println("--- Case 1: no exception ---");
        process(10, 2);

        System.out.println("--- Case 2: caught exception ---");
        process(10, 0);

        System.out.println("--- Case 3: uncaught exception ---");
        processWithoutCatch(10, 0);   // finally STILL runs, then the
                                      // exception propagates anyway (see
                                      // "Introduction to Exceptions")
    }

    private static void process(int a, int b) {
        try {
            System.out.println("Result: " + (a / b));
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
        } finally {
            System.out.println("finally: cleanup runs either way");
        }
    }

    private static void processWithoutCatch(int a, int b) {
        try {
            System.out.println("Result: " + (a / b));
        } finally {
            // try WITHOUT a catch, paired only with finally, is completely
            // legal -- finally still runs, then the exception (having no
            // catch to stop it) continues propagating exactly as if there
            // were no try block at all.
            System.out.println("finally: cleanup runs even without a catch block");
        }
    }
}
```

> `finally` is what a resource-cleanup pattern like closing a file or a database connection is built on -- `try`-with-resources (covered where it's actually used, in the "File Reading" lesson) is really `finally`-based cleanup that the compiler writes for you automatically, not a fundamentally different mechanism.

finally and return: A Subtle Interaction

`finally` running unconditionally has one genuinely surprising consequence: if `finally` ITSELF contains a `return` (or a `throw`), it silently overrides whatever the `try` or `catch` block was about to return, discarding it completely -- including a genuinely uncaught exception that was already propagating.

```
// A genuinely surprising, real Java behavior -- and exactly why "Best
// Practices" recommends never putting a return (or another throw) inside a
// finally block. If finally itself returns a value, that value SILENTLY
// REPLACES whatever the try or catch block was about to return -- and if
// finally throws, it SILENTLY REPLACES whatever exception was already
// propagating, discarding it completely.
public class FinallyOverridingReturnExample {
    public static void main(String[] args) {
        System.out.println("brokenDivide(10, 0) returned: " + brokenDivide(10, 0));
        // Prints -1, NOT the ArithmeticException you might expect --
        // finally's "return -1" swallowed the exception entirely. The
        // catch block never even got a chance to run its own return.
    }

    private static int brokenDivide(int a, int b) {
        try {
            return a / b;              // throws ArithmeticException when b == 0
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
            return -1;
        } finally {
            // This return DISCARDS the catch block's "return -1" (in this
            // case, coincidentally the same value) -- but more importantly,
            // it would ALSO discard a genuinely uncaught, unrelated
            // exception from anywhere in the try/catch, silently, with no
            // trace of it ever happening. That's the real danger, not the
            // duplicate value here.
            return -1;
        }
    }
}
```

> This isn't a bug or an edge case Java accidentally allows -- it's simply how `finally` running unconditionally interacts with `return` / `throw`. It's exactly why "Best Practices" recommends treating a `return` or `throw` inside `finally` as something to avoid, not a convenient shortcut.

Best Practices

- **Keep `try` blocks focused on the specific code that can actually fail** -- wrapping far more code than necessary makes it harder to tell which line an exception could realistically come from.
- **Order multiple `catch` blocks from most specific to most general** -- see the warning in "Multiple catch Blocks", and see "Exception Hierarchy" for how to reason about which type is more general.
- **Prefer multi-catch over duplicating identical `catch` bodies** -- see `MultiCatchExample` -- but only when the handling logic is genuinely identical, not just similar.

- **Never put a `return` or `throw` inside a `finally` block** -- see "finally and return: A Subtle Interaction" -- use `finally` purely for cleanup that doesn't affect what gets returned or thrown.

Common Mistakes

- **Ordering a superclass's `catch` block before a subclass's.** The compiler rejects this outright as unreachable code -- see the warning in "Multiple catch Blocks: Matching in Order".
- **Assuming `finally` doesn't run when a `try` block returns.** It still runs, between the `return` statement being evaluated and control actually leaving the method -- see `FinallyAlwaysRunsExample`.
- **Not realizing a `return` inside `finally` silently discards an in-flight exception.** No warning, no trace of the original problem -- see `FinallyOverridingReturnExample`.
- **Wrapping an entire method body in one giant `try` block "just in case."** This makes it much harder to tell, later, which specific line an exception is actually protecting against.

Summary, Cheat Sheet, and Glossary

`try` marks code that might fail; `catch` specifies what to do for a specific exception type if it does, checked top to bottom until one matches; multi-catch (`|`) lets one `catch` block handle several unrelated types with identical handling; `finally` runs unconditionally, in every case, making it the right place for cleanup -- but never for `return` or `throw`, since that silently overrides whatever the `try`/`catch` was about to produce.

Quick reference:

```
try {
    riskyOperation();
} catch (SpecificException e) {
    // handle the specific case
} catch (AnotherException | YetAnotherException e) {
    // multi-catch: identical handling for two unrelated types
} finally {
    // always runs -- cleanup only, never return/throw here
}
```

Glossary

try — A block marking code that might throw an exception.

catch — A block specifying how to handle a specific exception type thrown inside a preceding `try` block.

Multi-Catch — A single `catch` block, using `|`, that handles several unrelated exception types identically.

finally — A block that runs unconditionally after `try` / `catch`, regardless of whether an exception occurred or was caught.