Throw and Throws

The difference between the throw statement handing a Throwable instance to the JVM at runtime and the throws declaration being a purely compile-time contract -- fail-fast validation, rethrowing a caught exception as a different type with the cause parameter, and a checked exception propagating unhandled through a call chain. The 5th lesson in the Exception Handling series.

Beginner 20 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/throw-and-throws

---

You have already used `throw` and `throws` in passing across this series — a `throw` inside an example, a `throws IOException` on a method signature. This lesson stops to look at each of them on its own: what `throw` actually does when it executes, what `throws` declares and does not do, and how the two work together to move an exception from where it happens to where it gets handled.

What Are throw and Throws?

`throw` is a STATEMENT — it executes at a specific point in your code and immediately hands a `Throwable` instance to the JVM, right now, at runtime. `throws` is a DECLARATION — it appears in a method's signature and tells the compiler (and anyone reading the method) which checked exceptions might come out of it; writing `throws` does not, by itself, throw anything or run any code at all.

Why Does It Exist?

Without `throw`, code would have no way to signal "something went wrong" other than a magic return value (like `-1` or `null`) — the exact problem "Introduction to Exceptions" opened with. Without `throws`, a checked exception thrown deep inside a call chain would have no compiler-verified path back to whoever needs to handle it — every method in between could silently forget about it. Together, `throw` creates the failure and `throws` (for checked exceptions) makes sure the compiler tracks its path to a handler.

The throw Statement

`throw` takes a single `Throwable` instance — usually one you construct on the spot with `new` — and transfers control away from that point immediately. Any code written directly after an unconditional `throw` is UNREACHABLE and the compiler rejects it outright.

```
public class ThrowStatementBasicsExample {

    public static void main(String[] args) {
        try {
            reject();
        } catch (IllegalStateException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }

    static void reject() {
        // "throw" is a statement: it executes right now, immediately handing
        // a Throwable instance to the JVM. Execution of this method stops here.
        throw new IllegalStateException("this operation is not allowed right now");

        // Any code placed here would be UNREACHABLE and would not even compile
        // -- the compiler knows the throw above always transfers control away.
    }
}
```

In `reject()`, execution never gets past the `throw` line — there is nothing to "return" from, and there is nowhere to place code after it in that same block.

Throwing to Fail Fast

A very common use of `throw` is validating a method's arguments at the very top, before any real work happens, and throwing immediately if they're invalid. This is called "failing fast" — the failure is reported at its true origin, instead of surfacing later as a confusing symptom somewhere else.

```
public class FailFastValidationExample {

    public static void main(String[] args) {
        System.out.println(applyDiscount(100.0, 10));

        try {
            applyDiscount(100.0, -5);
        } catch (IllegalArgumentException e) {
            System.out.println("Rejected: " + e.getMessage());
        }
    }

    static double applyDiscount(double price, int percent) {
        // Fail fast: validate arguments first and throw immediately, before
        // any real work happens -- this keeps the rest of the method free of
        // defensive checks and makes the bad input's origin obvious.
        if (percent < 0 || percent > 100) {
            throw new IllegalArgumentException("percent must be between 0 and 100, was " +
percent);
        }

        return price - (price * percent / 100);
    }
}
```

`applyDiscount(...)` checks `percent` before doing any math — if you removed that check, an invalid percent wouldn't crash immediately; it would quietly produce a wrong price, a far harder bug to track down.

> When you write a `throw` for invalid input, put it as the very first thing the method does. A validation check buried in the middle of a method is easy to miss and easy to bypass by accident when the method is later refactored.

Rethrowing: Catching One Exception, Throwing Another

Sometimes the right response to catching an exception isn't to handle it, but to `throw` a different, more meaningful exception in its place — one described in terms of what YOUR method promises, not in terms of some internal detail it depends on.

```
public class RethrowingCaughtExceptionExample {

    public static void main(String[] args) {
        try {
            loadConfiguration("settings.cfg");
        } catch (IllegalStateException e) {
            System.out.println("Startup failed: " + e.getMessage());
        }
    }

    static void loadConfiguration(String path) {
        try {
            parse(path);
        } catch (NumberFormatException e) {
            // Rethrowing: catch a specific, low-level exception and throw a
            // different, higher-level one in its place -- the caller sees a
            // failure described in terms this method's contract, not in
            // terms of an internal detail (a malformed number) it never
            // promised to expose. The original is kept as the "cause" so
            // no diagnostic information is lost.
            throw new IllegalStateException("configuration file is corrupt: " + path, e);
        }
    }

    static void parse(String path) {
        Integer.parseInt("not-a-number");
    }
}
```

`loadConfiguration(...)` catches the low-level `NumberFormatException` from `parse(...)` and throws a `IllegalStateException` that actually means something to its own caller — "configuration file is corrupt" is meaningful outside this method; "a number failed to parse" is not. Notice the second constructor argument: passing the original exception as the `cause` keeps its message and stack trace attached, so nothing is lost by rethrowing a different type. You'll formalize this exact pattern — designing your own exception types instead of reusing built-in ones like `IllegalStateException` — in the next lesson.

The throws Declaration and Propagation

`throws` on a method signature does not run anything by itself — it only tells the compiler that a checked exception CAN come out of this method, without that method catching it. As you saw in "Checked vs Unchecked Exceptions," this is what makes the checked-exception contract enforceable: every method in a call chain must either `catch` the exception or add its own `throws`, all the way up.

```
import java.io.FileNotFoundException;

public class ThrowsDeclarationPropagationExample {

    public static void main(String[] args) {
        try {
            step1();
        } catch (FileNotFoundException e) {
            System.out.println("Caught at the top: " + e.getMessage());
        }
    }

    // Each method here declares "throws" instead of catching -- the checked
    // exception simply passes through, unhandled, all the way up the call
    // chain until something finally catches it in main. "throws" costs
    // nothing to write and does not stop execution by itself; it is a
    // compile-time declaration, not a runtime action like "throw".
    static void step1() throws FileNotFoundException {
        step2();
    }

    static void step2() throws FileNotFoundException {
        step3();
    }

    static void step3() throws FileNotFoundException {
        throw new FileNotFoundException("report.csv");
    }
}
```

Here, `step3()` is the only method with an actual `throw` — `step1()` and `step2()` never touch the `FileNotFoundException` at all, they just declare `throws FileNotFoundException` and let it pass straight through. Nothing runs differently in `step1()` or `step2()` because of that declaration; it's purely compile-time bookkeeping that keeps the compiler honest about where the exception can end up.

> Declaring `throws SomeException` does not catch or reduce the exception in any way — it only shifts the compiler's obligation to whoever calls this method. If nothing up the call chain ever `catch`es it, the exception still terminates the program when it's not caught, exactly like you saw in "Introduction to Exceptions."

throw vs throws: The Core Distinction

It's worth stating the difference directly, since the similar names invite confusion: `throw` is something a method's BODY does, at a specific line, at runtime, and it can only ever appear once per execution path leading to it. `throws` is something a method's SIGNATURE declares, at compile time, and a single method can declare as many exception types as it needs (comma-separated) without ever necessarily calling `throw` itself, as `step1()` and `step2()` demonstrate above.

Best Practices

- Validate arguments and throw at the very top of a method — fail fast, at the true source of the problem.
- When rethrowing a different exception type, always pass the original as the `cause` so no diagnostic information is lost.
- Only add `throws` for checked exceptions your method (or something it calls) can genuinely produce — don't declare it defensively "just in case."
- Keep the exception type you throw meaningful to the CALLER, not just accurate to your own implementation details.

Common Mistakes

- Writing code after an unconditional `throw` and being surprised the compiler rejects it as unreachable.
- Assuming a `throws` declaration on a method somehow "handles" the exception — it does not catch or suppress anything.
- Rethrowing a caught exception as a new type without passing the original as `cause`, silently discarding its stack trace.
- Throwing a generic, low-information exception (`throw new RuntimeException("error")`) instead of one whose type and message actually describe what went wrong.

Summary, Cheat Sheet, and Glossary

Summary

- `throw` is a runtime statement that immediately hands a `Throwable` instance to the JVM and transfers control away.
- `throws` is a compile-time declaration on a method signature that lets a checked exception propagate without being caught there.
- Failing fast means validating input and throwing at the very start of a method, at the true origin of the problem.
- Rethrowing a different, more meaningful exception type (with the original passed as `cause`) is a common and safe pattern.
- `throw` executes once per path at a specific line; `throws` can list multiple exception types and requires no `throw` in that same method.

Cheat Sheet

```
// throw: a runtime statement
void reject() {
    throw new IllegalStateException("not allowed");
}

// Fail fast
void applyDiscount(double price, int percent) {
    if (percent < 0 || percent > 100) {
        throw new IllegalArgumentException("percent must be between 0 and 100, was " +
percent);
    }
}

// Rethrowing with cause
try {
    parse(input);
} catch (NumberFormatException e) {
    throw new IllegalStateException("configuration file is corrupt", e);
}

// throws: a compile-time declaration, propagating without a catch
void step1() throws FileNotFoundException {
    step2(); // no catch here -- just passes through
}
```

Glossary

- **throw**: a statement that immediately hands a `Throwable` instance to the JVM, transferring control away.
- **throws**: a method-signature declaration listing checked exceptions the method may let propagate.
- **Fail fast**: throwing immediately upon detecting invalid input, at the true origin of the problem.
- **Rethrowing**: catching one exception and throwing a different one in its place, usually with the original passed as `cause`.
- **Propagation**: a checked exception passing unhandled through a chain of methods that each declare it with `throws`.