Creating and Throwing Custom Exceptions

Defining your own exception type by extending Exception (checked) or RuntimeException (unchecked) -- adding fields that carry extra context, mirroring Throwable's four standard constructors, and building your own small exception hierarchy with a shared custom base class. The 6th lesson in the Exception Handling series.

Beginner 20 min

Last updated: November 10, 2026
Read online: https://www.learnforgex.com/en/topics/custom-exceptions

---

Every exception you've thrown so far in this series has been a built-in Java type — `IllegalStateException`, `IllegalArgumentException`, `IOException`. Those work, but they can only ever say as much as their generic name and a text message allow. This lesson covers designing your OWN exception types: classes that describe a failure in the vocabulary of your own application, and that can carry more than a message can.

What Is a Custom Exception?

A custom exception is simply a class YOU write that extends `Exception` (making it checked) or `RuntimeException` (making it unchecked) — nothing more is required to make it a real, usable exception type. Once defined, it can be thrown, caught, and it participates in the exact same `catch` / `throws` /hierarchy rules you've already seen for built-in exceptions.

Why Does It Exist?

A generic exception like `IllegalStateException` tells a caller almost nothing about WHAT specifically went wrong beyond a string message that code can't reliably act on. A custom exception type gives a `catch` block something concrete to match against — `catch (InsufficientFundsException e)` is unambiguous in a way `catch (Exception e)` with a message check never is — and it can carry structured data (an invalid value, an error code) that a generic exception has no field for.

A Minimal Custom Exception

The simplest custom exception adds nothing beyond a constructor that forwards a message to its superclass.

```
public class BasicCustomExceptionExample {

    // A custom checked exception: extends Exception (not RuntimeException),
    // so every caller of a method that throws it must catch it or declare
    // it with "throws" -- the same compiler-enforced contract you saw with
    // built-in checked exceptions like IOException.
    static class InsufficientFundsException extends Exception {
        InsufficientFundsException(String message) {
            super(message);
        }
    }

    public static void main(String[] args) {
        try {
            withdraw(100.0, 250.0);
        } catch (InsufficientFundsException e) {
            System.out.println("Withdrawal failed: " + e.getMessage());
        }
    }

    static void withdraw(double balance, double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException(
                    "cannot withdraw " + amount + ", balance is only " + balance);
        }
        System.out.println("withdrew " + amount);
    }
}
```

`InsufficientFundsException` extends `Exception`, so it's checked — `withdraw(...)` must declare `throws InsufficientFundsException`, and `main` must either `catch` it or declare it too, exactly like you saw with built-in checked exceptions in "Checked vs Unchecked Exceptions."

> By convention, every custom exception's class name ends in `Exception` — `InsufficientFundsException`, not `InsufficientFunds`. This isn't compiler-enforced, but it's a strong readability convention that every Java codebase expects.

Carrying Extra Context

The real advantage of a custom exception over a built-in one is that it can hold FIELDS of its own — data a `catch` block can read back out, beyond just the message string.

```
public class CustomExceptionWithContextExample {

    // A custom unchecked exception: extends RuntimeException, so no
    // "throws" declaration is required at call sites. It carries an EXTRA
    // field beyond the inherited message -- the invalid value itself --
    // giving a catch block structured data to act on, not just text.
    static class InvalidOrderQuantityException extends RuntimeException {
        private final int quantity;

        InvalidOrderQuantityException(int quantity) {
            super("order quantity must be positive, was " + quantity);
            this.quantity = quantity;
        }

        int getQuantity() {
            return quantity;
        }
    }

    public static void main(String[] args) {
        try {
            placeOrder(-3);
        } catch (InvalidOrderQuantityException e) {
            System.out.println(e.getMessage());
            System.out.println("rejected quantity was: " + e.getQuantity());
        }
    }

    static void placeOrder(int quantity) {
        if (quantity <= 0) {
            throw new InvalidOrderQuantityException(quantity);
        }
        System.out.println("order placed for " + quantity);
    }
}
```

`InvalidOrderQuantityException` stores the rejected `quantity` in its own field, alongside the inherited message. The `catch` block in `main` calls `e.getQuantity()` to get that value back directly — no parsing a message string required.

Matching Built-in Constructor Shapes

`Throwable` itself offers four constructors: no-argument, message-only, message-with-cause, and cause-only. A well-designed custom exception commonly mirrors all four, so callers can use it exactly the way they already use built-in exceptions.

```
public class CustomExceptionConstructorsExample {

    // Mirroring the four constructors Throwable itself offers is a common
    // convention -- it lets callers of your exception choose exactly the
    // same shapes they already use with built-in exceptions: no details,
    // a message only, a message with a cause, or a cause only.
    static class ReportGenerationException extends RuntimeException {
        ReportGenerationException() {
            super();
        }

        ReportGenerationException(String message) {
            super(message);
        }

        ReportGenerationException(String message, Throwable cause) {
            super(message, cause);
        }

        ReportGenerationException(Throwable cause) {
            super(cause);
        }
    }

    public static void main(String[] args) {
        try {
            throw new ReportGenerationException("PDF renderer returned no pages");
        } catch (ReportGenerationException e) {
            System.out.println("message-only: " + e.getMessage());
        }

        try {
            Exception original = new IllegalStateException("renderer not initialized");
            throw new ReportGenerationException("report generation failed", original);
        } catch (ReportGenerationException e) {
            System.out.println("message+cause: " + e.getMessage() + " <- " + e.getCause());
        }
    }
}
```

`ReportGenerationException` simply forwards each constructor's arguments to `super(...)`. This is what makes wrapping — the pattern from "Throw and Throws," where a caught exception is rethrown as a different, more meaningful type — work smoothly with your own exception types too: the message-and-cause constructor is what you'd reach for there.

Building Your Own Exception Hierarchy

Just as `IOException` and `SQLException` both sit under the shared `Exception` type, your own exceptions can share a common custom base class — letting a caller choose between catching broadly (the shared problem) or narrowly (one specific cause).

```
public class CustomExceptionHierarchyExample {

    // A small hierarchy of your own, built the same way Java's own
    // exception classes are: one shared base type, several specific
    // subclasses. Code that only cares about "some payment problem
    // happened" can catch the base type; code that needs to react
    // differently to each cause can catch the specific subclasses instead.
    static class PaymentException extends RuntimeException {
        PaymentException(String message) {
            super(message);
        }
    }

    static class CardDeclinedException extends PaymentException {
        CardDeclinedException(String message) {
            super(message);
        }
    }

    static class PaymentGatewayTimeoutException extends PaymentException {
        PaymentGatewayTimeoutException(String message) {
            super(message);
        }
    }

    public static void main(String[] args) {
        // Catching by the shared base type, exactly like catching by
        // RuntimeException in "Exception Hierarchy" -- both subclasses
        // match, without listing each one.
        for (String scenario : new String[] {"declined", "timeout"}) {
            try {
                charge(scenario);
            } catch (PaymentException e) {
                System.out.println("payment failed (" + e.getClass().getSimpleName() + "):
" + e.getMessage());
            }
        }
    }

    static void charge(String scenario) {
        if (scenario.equals("declined")) {
            throw new CardDeclinedException("card was declined by the issuer");
        }
        throw new PaymentGatewayTimeoutException("gateway did not respond in time");
    }
}
```

Expected Output

```
payment failed (CardDeclinedException): card was declined by the issuer
payment failed (PaymentGatewayTimeoutException): gateway did not respond in time
```

`CardDeclinedException` and `PaymentGatewayTimeoutException` both extend `PaymentException`. The loop in `main` catches only `PaymentException` and handles both concrete failures with one `catch` block — the same polymorphic matching by supertype you saw with built-in types in "Exception Hierarchy," now applied to a hierarchy you designed yourself.

> Don't build a deep hierarchy of custom exceptions "just in case" future subtypes might be useful. Start with one exception type per genuinely distinct failure your code needs to react to differently — add a subclass later, when a real need for one actually appears.

Best Practices

- Name every custom exception type with an `Exception` suffix, matching the convention every Java codebase expects.
- Extend `RuntimeException` unless callers can genuinely recover from the failure and should be forced to handle it — the same guideline from "Checked vs Unchecked Exceptions" applies to your own types too.
- Add fields for any structured data a `catch` block might need, instead of encoding it only into the message string.
- Mirror `Throwable`'s standard constructors (no-arg, message, message+cause, cause) so your exception composes cleanly with wrapping.

Common Mistakes

- Extending `Throwable` directly instead of `Exception` or `RuntimeException` — this bypasses the checked/unchecked distinction entirely and is almost never what you want.
- Forgetting to call `super(message)` (or `super(message, cause)`), leaving `getMessage()` returning `null` for no reason.
- Designing a custom exception hierarchy that's deeper or broader than any code actually needs to catch differently.
- Reaching for a custom exception when a built-in one (like `IllegalArgumentException`) already says exactly what's needed — not every failure needs a brand-new type.

Summary, Cheat Sheet, and Glossary

Summary

- A custom exception is a class you write extending `Exception` (checked) or `RuntimeException` (unchecked).
- Custom exceptions let a `catch` block match precisely and carry structured data beyond a message string.
- By convention, custom exception class names always end in `Exception`.

- Mirroring `Throwable`'s four standard constructors keeps a custom exception compatible with wrapping.
- A shared custom base class lets callers catch broadly or narrowly, the same polymorphic matching built-in hierarchies use.

Cheat Sheet

```
// Minimal checked custom exception
class InsufficientFundsException extends Exception {
    InsufficientFundsException(String message) {
        super(message);
    }
}

// Unchecked, with extra context
class InvalidOrderQuantityException extends RuntimeException {
    private final int quantity;
    InvalidOrderQuantityException(int quantity) {
        super("invalid quantity: " + quantity);
        this.quantity = quantity;
    }
    int getQuantity() { return quantity; }
}

// A small hierarchy of your own
class PaymentException extends RuntimeException {
    PaymentException(String message) { super(message); }
}
class CardDeclinedException extends PaymentException {
    CardDeclinedException(String message) { super(message); }
}
```

Glossary

- **Custom exception**: a class you define that extends `Exception` or `RuntimeException`.
- **Context (in an exception)**: extra fields on a custom exception, beyond the inherited message, that a `catch` block can read.
- **Constructor mirroring**: giving a custom exception the same four constructor shapes as `Throwable`.
- **Custom hierarchy**: a shared base exception type with your own subclasses under it, catchable broadly or narrowly.