Checked vs Unchecked Exceptions

The difference between a checked exception's compiler-enforced throws/catch contract and an unchecked exception's (the RuntimeException family) complete lack of that requirement -- when to choose which, wrapping a checked exception as unchecked with the cause parameter, and the restriction that an overriding method's throws declaration can only be NARROWED. The 4th lesson in the Exception Handling series.

Beginner 20 min

Last updated: November 9, 2026
Read online: https://www.learnforgex.com/en/topics/checked-vs-unchecked-exceptions

---

In "Exception Hierarchy" you saw that `Exception` splits into two broad branches: `RuntimeException` and everything that is NOT `RuntimeException`. That split carries a much more concrete consequence than its name suggests — a rule the compiler genuinely enforces. This lesson covers exactly what that rule is, why it exists, and how to choose between the two in professional Java code.

What Are Checked and Unchecked Exceptions?

A **checked exception** is any class under `Exception` that is NOT `RuntimeException` (like `IOException`, `SQLException`). If a method can throw a checked exception, it MUST declare that with `throws` in its signature — and every piece of code that calls it MUST either `catch` it or declare it with `throws` in its own signature too. An **unchecked exception** is `RuntimeException` itself (and `Error`) along with all of their subclasses — neither declaring nor catching them is required; the compiler demands nothing.

Why Does It Exist?

The point of this distinction is to let an API offer its callers two different contracts. A checked exception says: "this operation can fail, that failure is OUTSIDE YOUR CONTROL, but it's a reasonably expected condition — you cannot ignore it" (a missing file, a dropped network connection). An unchecked exception usually represents a PROGRAMMING ERROR (dereferencing a null reference, accessing an invalid index) — forcing every call site to `catch` that would make code unreadable and would hide the actual bug in the source rather than surfacing it.

History

The checked-exception mechanism has existed since Java 1.0 (1996) — it's widely regarded as one of the language's most debated design decisions. Languages like C++ never made exceptions mandatory at all, but Java's designers deliberately chose checked exceptions to improve reliability. Over time, the community realized checked exceptions weren't the right tool for EVERY situation — even Java's own standard library eventually added `RuntimeException`-based alternatives (like `java.io.UncheckedIOException`, added in Java 8).

The Compiler-Enforced Contract: Checked Exceptions

A method that can throw a checked exception must explicitly declare it with `throws` — and EVERY caller of that method must either handle it with a `catch` block or add it to their own `throws` declaration. This isn't an optional suggestion; it's a rule enforced at compile time.

```
import java.io.IOException;

// A checked exception is anything that extends Exception but NOT RuntimeException
// (see "Exception Hierarchy" for that split). The compiler enforces a real contract
// around it: if a method can throw one, every caller must either catch it or declare
// it with `throws` in its own signature -- there is no way to call this method and
// silently ignore that possibility, the way you can with an unchecked exception.
public class CheckedExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            System.out.println("Loaded: " + readSetting("timeout"));
            System.out.println("Loaded: " + readSetting("missing"));
        } catch (IOException e) {
            // Forced by the compiler -- removing this catch (or a `throws IOException`
            // on main) would not compile.
            System.out.println("Caught checked exception: " + e.getMessage());
        }
    }

    // IOException is used here purely as a familiar, real checked exception to
    // demonstrate the COMPILER'S enforcement -- no actual file I/O happens (see the
    // File Reading lesson for real IOException scenarios from disk access).
    private static String readSetting(String key) throws IOException {
        if (key.equals("missing")) {
            throw new IOException("setting not found: " + key);
        }
        return "30s";
    }
}
```

In this example, `readSetting(...)` declares `throws IOException`, so `main` must either `catch` it or declare `throws IOException` itself — there's no third option, the compiler doesn't allow it.

Unchecked Exceptions: The RuntimeException Family

`RuntimeException` (and its subclasses) carries the exact opposite contract: no `throws` declaration or `catch` block is REQUIRED at all. This is the shared trait of classes like `ArithmeticException` and `NumberFormatException`, which you saw in "Exception Hierarchy".

```
// An unchecked exception is any RuntimeException (or subclass of it). The compiler
// places NO obligation on a caller to catch it or declare it -- this file compiles
// and runs perfectly fine with no try/catch anywhere and no `throws` on main, even
// though divide(...) can clearly fail.
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        System.out.println("Before calling divide(...)");
        int result = divide(10, 0);
        System.out.println("This line never runs: " + result);
    }

    private static int divide(int a, int b) {
        return a / b; // throws ArithmeticException -- unchecked, no `throws` clause needed
    }
}
```

This example compiles and runs with no `try` / `catch` or `throws` declaration whatsoever — the fact that `divide(...)` can throw an `ArithmeticException` makes no difference to the compiler at all.

> The fastest way to tell whether an exception is checked or unchecked: check whether the class extends `RuntimeException` (or `Error`) — if it does, it's unchecked; if it doesn't (and it's under `Exception`), it's checked.

When to Use Which?

The generally accepted guideline: if the caller can reasonably RECOVER from the condition (a missing file — asking the user for a different path is a plausible response), a checked exception makes sense. If the condition represents a PROGRAMMING ERROR (an invalid argument, a null reference) or something the caller genuinely can't do anything about, an unchecked exception is the better fit. Most modern Java libraries (Spring included) deliberately limit their use of checked exceptions, because the mandatory `catch` / `throws` chain at every API boundary can quickly bloat a codebase.

Wrapping a Checked Exception as Unchecked

Sometimes the calling code's signature can't declare a checked exception (for example, when overriding an interface method — you'll see this in the next section) or such a declaration would needlessly complicate the API. In that case, catching the checked exception and WRAPPING it inside an unchecked one is a common solution.

```
import java.io.IOException;

// A common, real pattern: catch a checked exception and re-throw it wrapped inside
// an unchecked one, when the surrounding method's signature can't (or shouldn't)
// declare `throws IOException` itself -- for example, because it overrides an
// interface method that doesn't declare it (see the next example in this lesson).
public class WrappingCheckedAsUncheckedExample {
    public static void main(String[] args) {
        loadRequiredSetting("missing");
    }

    // No `throws` clause needed here -- RuntimeException is unchecked, so this
    // compiles even though readSetting(...) itself can throw a checked exception.
    private static void loadRequiredSetting(String key) {
        try {
            readSetting(key);
        } catch (IOException e) {
            // The constructor's second argument is the ORIGINAL exception, passed
            // as the "cause" -- nothing about the checked exception's message or
            // stack trace is lost, it's just no longer something the caller is
            // FORCED to handle.
            throw new RuntimeException("failed to load required setting: " + key, e);
        }
    }

    private static String readSetting(String key) throws IOException {
        if (key.equals("missing")) {
            throw new IOException("setting not found: " + key);
        }
        return "30s";
    }
}
```

Here, the `IOException` is wrapped using the second argument of `RuntimeException`'s constructor, the `cause` parameter — the original exception's information (message, stack trace) isn't lost, it's just no longer something the caller is FORCED to handle.

> When wrapping a checked exception, don't forget to pass the original exception as the `cause` (like `new RuntimeException(message, e)`) — otherwise the underlying error's stack trace is lost, making debugging much harder.

The throws Restriction on Overridden Methods

When overriding a method, you may declare FEWER (or a NARROWER subtype of) the checked exceptions the superclass/interface declared — but you may never declare MORE, or a BROADER checked type. This rule is enforced by the compiler.

```
import java.io.FileNotFoundException;
import java.io.IOException;

// When overriding a method, you may declare FEWER (or NARROWER) checked exceptions
// than the method you're overriding -- but never MORE, and never a BROADER checked
// exception type. This is enforced by the compiler, not just a convention.
public class OverridingThrowsRestrictionExample {
    interface SettingsSource {
        String read(String key) throws IOException;
    }

    // Narrows IOException down to FileNotFoundException (one of its subclasses) --
    // legal, because any caller that already handles IOException also handles this.
    static class StrictSettingsSource implements SettingsSource {
        @Override
        public String read(String key) throws FileNotFoundException {
            if (key.equals("missing")) {
                throw new FileNotFoundException("setting not found: " + key);
            }
            return "30s";
        }
    }

    // Drops the `throws` clause entirely -- also legal, since declaring nothing is
    // narrower than declaring IOException.
    static class InMemorySettingsSource implements SettingsSource {
        @Override
        public String read(String key) {
            return "30s";
        }
    }

    public static void main(String[] args) throws IOException {
        SettingsSource source = new StrictSettingsSource();
        System.out.println(source.read("timeout"));

        SettingsSource memorySource = new InMemorySettingsSource();
        System.out.println(memorySource.read("timeout"));

        // If StrictSettingsSource.read tried to declare `throws Exception` (broader
        // than IOException), this file would NOT compile -- the restriction is
        // checked by the compiler at the override site, not at any call site.
    }
}
```

Expected Output

```
30s
30s
```

In this example, `StrictSettingsSource` NARROWS the interface's declared `IOException` down to its subclass `FileNotFoundException`; `InMemorySettingsSource` DROPS the `throws` declaration entirely — both are valid, because both are a subset of the `IOException` contract the caller already expects.

Best Practices

- Use checked exceptions only for conditions the caller can genuinely recover from; prefer `RuntimeException` for programming errors.
- When wrapping a checked exception, always pass the original as the `cause` — never lose that information.
- Justify checked-exception usage when designing an API — every checked exception imposes a `catch` / `throws` burden on EVERYONE who calls it.
- If you forget which checked exceptions an overriding method is allowed to declare, the compiler will stop you — but knowing the restriction upfront helps you design interfaces correctly the first time.

Common Mistakes

- Leaving every checked exception in an empty `catch (IOException e) {}` block — an anti-pattern you'll keep seeing across these lessons, one that silently swallows the error entirely.
- Catching checked exceptions with an overly broad type like `catch (Exception e)` without thinking — this lumps both checked and unchecked exceptions into the same block, intentionally or not.
- Forgetting to pass the `cause` parameter when wrapping an exception — the original error's stack trace is lost.
- Assuming checked exceptions are always "better" or "more professional" — in practice, modern Java tends to favor unchecked exceptions, reserving checked exceptions for conditions that are genuinely recoverable.

Summary, Cheat Sheet, and Glossary

Summary

- Checked exceptions are `Exception` subclasses outside `RuntimeException` — declaring `throws` or catching them is MANDATORY.
- Unchecked exceptions are `RuntimeException` (and `Error`) and their subclasses — no declaration is mandatory.
- Checked exceptions typically represent recoverable, external conditions; unchecked exceptions typically represent programming errors.
- Wrapping a checked exception inside an unchecked one, with the original passed as `cause`, is a common and safe technique.
- An overriding method may only declare FEWER or a NARROWER subtype of the checked exceptions its superclass declared.

Cheat Sheet

```
// Checked  -- declaring throws / catching is MANDATORY
void readFile() throws IOException { ... }

// Unchecked -- no declaration required
void divide(int a, int b) { return a / b; }

// Wrapping
try {
    readFile();
} catch (IOException e) {
    throw new RuntimeException("failed", e); // don't forget the cause
}

// Override restriction: you can only NARROW
interface Source { String read() throws IOException; }
class Strict implements Source {
    public String read() throws FileNotFoundException { ... } // OK, a subtype
}
```

Glossary

- **Checked exception**: an `Exception` subclass outside `RuntimeException`; `throws`/`catch` is mandatory.
- **Unchecked exception**: `RuntimeException` (or `Error`) and its subclasses; no declaration is mandatory.
- **Wrapping**: catching one exception and re-throwing it as the `cause` of another.
- **cause**: an exception's reference to the original exception that led to it.
- **Narrowing**: an overriding method declaring a narrower subtype (or none) of the checked exception its superclass declared.