Module 12 of 20

Exceptions

The hierarchy, checked versus unchecked, try/catch/finally semantics, try-with-resources, custom exceptions with chaining, and handling practices.

Exceptions are how Java separates the normal path from failure. The mechanism is precise — a class hierarchy, compile-time checking for some of it, exact rules for finally and for resource cleanup — and the practices around it are what separate code that fails loudly and diagnosably from code that fails silently in production.

This module covers the Throwable hierarchy and how to read a stack trace; the checked/unchecked rules, throws propagation and wrapping; the semantics of try, catch and finally including the return and double-throw traps; try-with-resources with AutoCloseable and suppressed exceptions; designing custom exceptions with data and causes; and the handling rules — never swallow, catch narrowly, log once, fail fast — that teams enforce in review.

By the end you know exactly what happens when an exception is thrown, write handlers that preserve the evidence, and design exceptions for a layered application.

CONTENTS

1	The exception hierarchy and how to read a stack trace	13 min
2	Checked versus unchecked — the rules and the judgement	13 min
3	try, catch and finally — the exact semantics	14 min
4	try-with-resources and AutoCloseable	12 min
5	Custom exceptions and exception chaining	12 min
6	Exception handling that survives production	13 min
✓	Checkpoint — Exceptions	24 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

The exception hierarchy and how to read a stack trace

When something goes wrong at run time — an index out of range, a file missing, a null dereferenced — Java creates an **exception object** describing the failure and unwinds the call stack until something catches it. Every exception is an instance of a class in one hierarchy rooted at `Throwable`, and where a class sits in that hierarchy decides whether the compiler forces you to handle it. This lesson is the map of that hierarchy, the exceptions you will meet daily, and how to read the stack trace you get when one escapes.

The hierarchy

```
Throwable
├── Error                       — the JVM is in trouble; do not catch
│   ├── OutOfMemoryError
│   ├── StackOverflowError
│   ├── NoClassDefFoundError
│   └── AssertionError
└── Exception                   — something a program might recover from
    ├── IOException             — CHECKED: must be declared or caught
    │   ├── FileNotFoundException
    │   └── UncheckedIOException (RuntimeException, despite the name's neighbo
    ├── SQLException             — checked
    ├── InterruptedException     — checked
    ├── ReflectiveOperationException (ClassNotFoundException, …) — checked
    └── RuntimeException         — UNCHECKED: may be thrown anywhere without d
        ├── NullPointerException
        ├── IllegalArgumentException
        │   └── NumberFormatException
        ├── IllegalStateException
        ├── IndexOutOfBoundsException
        │   ├── ArrayIndexOutOfBoundsException
        │   └── StringIndexOutOfBoundsException
        ├── ArithmeticException
        ├── ClassCastException
        ├── UnsupportedOperationException
        ├── ConcurrentModificationException
        ├── NoSuchElementException
        └── DateTimeException
```

Three tiers to remember:

> `Error`: the JVM or the platform failed — memory exhausted, stack blown, a class missing at load time. Programs should not catch these (what would you do?), and catching `Throwable` to "be safe" swallows them.
>
> **Checked exceptions** (`Exception` and subclasses that are *not* `RuntimeException`): conditions a correct program should anticipate — a file may be absent, a network may drop. The compiler forces every caller to catch or declare them.
>
> **Unchecked exceptions** (`RuntimeException` and subclasses): programming errors and precondition violations — null where an object was required, an index past the end, an argument out of range. No declaration required; they propagate freely.

The next lesson is about the checked/unchecked decision; here, the key is that the *class* of an exception is the *category* of failure.

What an exception carries

```
Throwable t;
t.getMessage()        // the detail message given at construction, or null
t.getCause()          // the exception that caused this one, or null (chaining
t.getStackTrace()     // StackTraceElement[] — the frames at the point of crea
t.printStackTrace()   // to System.err; loggers do this properly
t.getSuppressed()     // exceptions suppressed during try-with-resources clear
t.toString()          // "java.lang.IllegalArgumentException: the message"
```

The stack trace is captured **when the object is constructed** — not when it is thrown. That is why `new Exception()` is expensive (filling the trace walks the stack) and why creating exceptions for control flow is slow (lesson 6).

Reading a stack trace

```
Exception in thread "main" java.lang.NumberFormatException: For input string:
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatEx
    at java.base/java.lang.Integer.parseInt(Integer.java:661)
    at java.base/java.lang.Integer.parseInt(Integer.java:777)
    at com.example.Parser.readAge(Parser.java:42)
    at com.example.Parser.parse(Parser.java:18)
    at com.example.Main.main(Main.java:9)
```

Read it this way:

The first line: the thread, the exception class, the message. Here the message tells you the bad input exactly.

The frames, top to bottom: innermost first. The top frames are usually library code that *detected* the problem; scan down to the first frame in **your** package — `Parser.readAge(Parser.java:42)` — that is where to look. The frames below show how execution got there.

`Caused by:` sections (chained exceptions) follow, each with its own trace, and `... 12 more` means frames identical to the enclosing trace were omitted.

A stack trace is a gift: it names the exact line. Learn to jump to the first frame of your own code rather than reading from the top.

The common unchecked exceptions, and what they mean

Exception	What it says about your code
`NullPointerException`	A null reference was dereferenced. Since Java 14 the message says *which* one: `Cannot invoke "String.length()" because "name" is null`.
`ArrayIndexOutOfBoundsException`	An index outside `0..length-1`; the message gives index and length.
`StringIndexOutOfBoundsException`	`charAt` / `substring` out of range.
`IllegalArgumentException`	A method received an argument it cannot accept — thrown by *you* in validation.
`IllegalStateException`	The object was not in a state that allows the call (`Iterator.remove` twice, a closed stream).
`NumberFormatException`	`parseInt` / `parseDouble` on non-numeric text.
`ArithmeticException`	Integer division by zero; `BigDecimal` non-terminating division.
`ClassCastException`	A downcast to a class the object is not.
`UnsupportedOperationException`	An optional operation not supported by this implementation (`List.of(…).add`).
`ConcurrentModificationException`	A collection was modified during iteration.
`NoSuchElementException`	`next()` past the end, `Optional.get()` on empty, `Scanner` with no token.

Most of these mean **a bug in the calling code**, not a condition to catch: fix the null, the index, the argument. The exception is the diagnosis.

`Throwable` versus `Exception` versus `RuntimeException` in a `catch`

`catch (Exception e)` catches every checked and unchecked exception but not `Error`s; `catch (RuntimeException e)` only unchecked; `catch (Throwable t)` everything including `OutOfMemoryError`. The last belongs only at the very top of a thread (a framework's request loop) where the intent is to log and keep the process alive — and even there, `Error`s are usually rethrown.

Exceptions are objects

You can store them, pass them, inspect them, and construct them without throwing. `throw` needs a `Throwable` expression: `throw new IllegalStateException("closed")`. The `throw` statement ends the method's normal execution immediately; code after it in the block is unreachable.

Interview angle

"Difference between `Error` and `Exception`?" `Error`s are JVM-level failures you do not catch; `Exception`s are conditions a program may handle.

"Checked vs unchecked — which classes?" Unchecked = `RuntimeException` and its subclasses (and `Error`s); everything else under `Exception` is checked.

"Is `NullPointerException` checked?" No — it extends `RuntimeException`.

"When is the stack trace captured?" At construction, in `Throwable`'s constructor.

"What does `... 12 more` mean?" Frames shared with the enclosing trace were elided.

Key takeaways

`Throwable` → `Error` (do not catch) and `Exception` → checked (compiler-enforced) vs `RuntimeException` (unchecked).

Unchecked exceptions usually mean a bug: fix the cause rather than catching.

Read a stack trace by finding the first frame in your own code; the message often names the bad value.

An exception is an object with a message, a cause and a trace captured at construction.

EXERCISE 1

Name that exception

Read an integer `n` and `n` commands, each of which triggers a runtime exception: `parse <text>` (`Integer.parseInt`), `index <i>` (index into a 3-element array), `div <a> <b>` (integer division), `cast` (cast a `String` held in an `Object` to `Integer`), `nullcall` (call `length()` on a null `String`). Catch each exception as `RuntimeException` and print `<simple class name>: <message>`; if no exception occurs print `ok <result>`.

Example: `3` then `parse 42`, `parse x`, `div 1 0` →

```
ok 42
NumberFormatException: For input string: "x"
ArithmeticException: / by zero
```

INPUT 1

```
3
parse 42
parse x
div 1 0
```

OUTPUT 1

```
ok 42
NumberFormatException: For input string: "x"
ArithmeticException: / by zero
```

INPUT 2

```
2
index 3
index 1
```

OUTPUT 2

```
ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
ok 20
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] arr = {10, 20, 30};
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            try {
                Object result;
                switch (cmd) {
                    case "parse" -> result = Integer.parseInt(in.next());
                    case "index" -> result = arr[in.nextInt()];
                    case "div" -> { int a = in.nextInt(); int b = in.nextInt(); resu
                    case "cast" -> { Object o = "text"; result = (Integer) o; }
                    default -> { String s = null; result = s.length(); }
                }
                System.out.println("ok " + result);
            } catch (RuntimeException e) {
                System.out.println(e.getClass().getSimpleName() + ": " + e.getMessag
            }
        }
    }
}
```

EXERCISE 2

Catch the specific one first

Read `n` commands. `parse <text>` calls `Integer.parseInt`; `divide <a> <b>` computes `a / b`; `index <i>` reads element `i` of a three-element array `{10, 20, 30}`. Run each inside one `try` with catch blocks ordered **most specific first**: `NumberFormatException` → `bad number: <text>`, `ArithmeticException` → `arithmetic: <message>`, `ArrayIndexOutOfBoundsException` → `index: <message>`, and a final `RuntimeException` → `other: <simple class name>`. On success print the result.

Example: `4` then `parse 42`, `parse x`, `divide 1 0`, `index 5` →

```
42
bad number: x
arithmetic: / by zero
index: Index 5 out of bounds for length 3
```

INPUT 1

```
4
parse 42
parse x
divide 1 0
index 5
```

OUTPUT 1

```
42
bad number: x
arithmetic: / by zero
index: Index 5 out of bounds for length
3
```

INPUT 2

```
2
index 2
divide 9 3
```

OUTPUT 2

```
30
3
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] data = {10, 20, 30};
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            String text = null;
            try {
                switch (cmd) {
                    case "parse" -> { text = in.next(); System.out.println(Integer.p
                    case "divide" -> { int a = in.nextInt(), b = in.nextInt(); Syste
                    default -> System.out.println(data[in.nextInt()]);
                }
            } catch (NumberFormatException e) {
                System.out.println("bad number: " + text);
            } catch (ArithmeticException e) {
                System.out.println("arithmetic: " + e.getMessage());
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("index: " + e.getMessage());
            } catch (RuntimeException e) {
                System.out.println("other: " + e.getClass().getSimpleName());
            }
        }
    }
}
```

QUIZ

1. Which of these is a checked exception?

A. `NullPointerException`

B. `IOException`

C. `IllegalArgumentException`

D. `OutOfMemoryError`

2. When is an exception's stack trace captured?

A. When it is thrown

B. When it is caught

C. When the exception object is constructed

D. When `printStackTrace` is called

3. Reading a stack trace, the most useful frame is usually…

A. The top one

B. The first frame in your own code, reading down from the top

C. The bottom one

D. The `Caused by` line only

4. `catch (Exception e)` does **not** catch…

A. `RuntimeException`

B. `IOException`

C. `StackOverflowError`

D. `NumberFormatException`

5. A `NullPointerException` in production most often means…

A. The network failed

B. A bug: a reference that should have been set was not — fix the code rather than catch it

C. The JVM is out of memory

D. A checked exception was ignored

ANSWER KEY · THE EXCEPTION HIERARCHY AND HOW TO READ A STACK TRACE

1. B `IOException` extends `Exception` directly. The first and third extend `RuntimeException`; the last is an `Error`.

2. C `Throwable`'s constructor walks the stack — which is why creating exceptions is expensive and rethrowing the same object keeps the original trace.

3. B Top frames are often library code that detected the problem; the first frame in your package is where to look.

4. C `Error`s are not `Exception`s. Only `catch (Throwable t)` would, and it should not.

5. B Unchecked exceptions signal programming errors; the message since Java 14 names the null variable.

LESSON 2 · 13 MIN

Checked versus unchecked — the rules and the judgement

Java is almost alone among mainstream languages in having **checked exceptions**: failures the compiler forces you to acknowledge at every call site. Whether that was a good idea is a twenty-year argument; that you must understand the rules is not. This lesson covers what the compiler demands, how `throws` propagates, how to convert between the two kinds, and the guidelines the JDK and *Effective Java* use to decide which kind a new exception should be.

The rule

If a method can throw a checked exception — one that is an `Exception` but not a `RuntimeException` — it must either **catch** it or **declare** it with `throws`:

```
static String read(Path p) throws IOException {        // declared: the caller
    return Files.readString(p);
}

static String readOrEmpty(Path p) {                    // caught: the caller
    try {
        return Files.readString(p);
    } catch (IOException e) {
        return "";
    }
}

static String broken(Path p) {
    return Files.readString(p);      // compile error: unreported exception IO
}
```

`throws` lists checked exceptions a method may let escape; unchecked ones may be listed for documentation but need not be. A method that declares `throws Exception` forces every caller to handle `Exception` — which is why broad declarations are a smell outside `main` and test code.

Propagation

An uncaught exception unwinds the stack: the current method ends, its caller's `catch` blocks are searched, then the caller's caller, until a handler is found or the thread dies (printing the trace). Each method on that path must have declared the checked exception. The chain of `throws` is the compile-time shadow of that run-time unwinding.

Unchecked exceptions unwind the same way — the difference is purely that the compiler does not track them.

Overriding and `throws`

An override may throw the same, fewer or narrower checked exceptions than the overridden method — never broader or new ones (Module 8). Consequence: an interface method that declares nothing (`Runnable.run`, `Function.apply`) can never be implemented by code that lets a checked exception escape. This is the source of most friction between checked exceptions and lambdas.

Converting checked to unchecked

When a checked exception cannot happen (a hard-coded `URI`), should not be recoverable (a config file that must exist), or must cross a boundary that does not allow it (a lambda), wrap it:

```
try {
    return Files.readString(path);
} catch (IOException e) {
    throw new UncheckedIOException("cannot read " + path, e);     // keep the
}

catch (InterruptedException e) {
    Thread.currentThread().interrupt();                            // restore
    throw new IllegalStateException("interrupted", e);
}
```

`UncheckedIOException` exists for exactly this; for other checked exceptions, `IllegalStateException(msg, cause)` or a custom unchecked exception. **Always pass the original as the cause** — losing it loses the real stack trace.

Converting the other way (throwing a checked exception from a place that caught an unchecked one) is rare and usually wrong.

The `throws` clause on `main`

`public static void main(String[] args) throws Exception` is the accepted way to let everything propagate in small programs and exercises: the JVM prints the trace and exits non-zero. In real applications `main` catches, logs and exits deliberately.

The design guideline: which kind?

The JDK's rule of thumb, refined by *Effective Java* items 70–71:

Checked when the condition is *expected* in a correct program and the caller can plausibly *recover*: a file that may not exist, a network that may fail, a user-entered date that may be malformed (`DateTimeParseException` is unchecked, though — the JDK is inconsistent). The compiler-enforced acknowledgement is the feature.

Unchecked when the condition is a *programming error* the caller cannot recover from at that point: null arguments, bad indices, illegal state, violated preconditions. Forcing every caller to `catch (NullPointerException)` would be noise; the fix is in the code, not the handler.

Modern practice leans unchecked for almost everything an application defines: frameworks (Spring, Jakarta) wrap data-access exceptions unchecked; Kotlin and Scala dropped checked exceptions entirely. The argument against checked: they leak implementation details up the stack (`throws SQLException` on a business method), they do not compose with lambdas and streams, and in practice most catch blocks just wrap and rethrow. The argument for: they document failure modes in the type system, and `IOException` genuinely should not be forgotten. A reasonable position: **use checked exceptions for recoverable conditions at API boundaries you control, unchecked for everything else, and never throw `Exception` or `Throwable`**.

Declaring precisely

```
void load() throws FileNotFoundException, ParseException     // specific: cal
void load() throws Exception                                  // lazy: caller
```

Declare the most specific types. If a method throws three unrelated checked exceptions, consider whether it is doing three things, or whether one domain exception should wrap them.

Checked exceptions and lambdas

```
paths.stream().map(p → Files.readString(p))     // compile error: IOExceptio
```

Options: catch and wrap inside the lambda (`UncheckedIOException`); a helper `static String readUnchecked(Path p)`; or a custom `ThrowingFunction` interface with an adapter. The first two are the everyday answers. Streams and checked exceptions do not mix; design library methods called from lambdas to throw unchecked.

Interview angle

"What is a checked exception?" A subclass of `Exception` but not `RuntimeException`; the compiler requires catch-or-declare.

"Why does `Files.readString` need a `try` but `Integer.parseInt` does not?" `IOException` is checked; `NumberFormatException` is unchecked.

"How do you throw a checked exception from a lambda?" You cannot let it escape; wrap it in an unchecked exception with the cause.

"When would you create a checked exception?" For a recoverable, expected condition at an API boundary — rarely, in modern code.

"Can an override add `throws IOException` if the parent declares none?" No.

Key takeaways

Checked = `Exception` minus `RuntimeException`; callers must catch or declare. Unchecked propagates silently.

`throws` propagates the obligation up the call chain; overrides may narrow, never broaden.

Wrap checked in unchecked (`UncheckedIOException`, `IllegalStateException`) at boundaries and in lambdas — always with the cause.

Checked for recoverable, expected conditions; unchecked for programming errors — and modern code leans unchecked.

Declare specific types; never `throws Exception` in an API.

EXERCISE 1

Declare, catch or wrap

Write a **checked** exception `class QuotaExceededException extends Exception` with a message. Write `static void consume(int used, int limit) throws QuotaExceededException` that throws when `used > limit`. Then write `static void consumeUnchecked(int used, int limit)` that calls `consume` and wraps the checked exception in an `IllegalStateException` **with the cause**.

Read an integer `limit`, an integer `n`, and `n` usage values. For each value call `consumeUnchecked`; catch `IllegalStateException` and print `wrapped: <cause class simple name>: <cause message>`, otherwise print `ok <used>`.

Example: `10`, `2`, `5 12` →

```
ok 5
wrapped: QuotaExceededException: used 12 of 10
```

INPUT 1

```
10
2
5 12
```

OUTPUT 1

```
ok 5
wrapped: QuotaExceededException: used 12
of 10
```

INPUT 2

```
0
3
0 1 0
```

OUTPUT 2

```
ok 0
wrapped: QuotaExceededException: used 1
of 0
ok 0
```

REFERENCE SOLUTION

```java
import java.util.*;

class QuotaExceededException extends Exception {
    QuotaExceededException(String message) { super(message); }
}

public class Main {
    static void consume(int used, int limit) throws QuotaExceededException {
        if (used > limit) throw new QuotaExceededException("used " + used + " of " +
    }

    static void consumeUnchecked(int used, int limit) {
        try {
            consume(used, limit);
        } catch (QuotaExceededException e) {
            throw new IllegalStateException("quota", e);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int limit = in.nextInt();
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int used = in.nextInt();
            try {
                consumeUnchecked(used, limit);
                System.out.println("ok " + used);
            } catch (IllegalStateException e) {
                Throwable cause = e.getCause();
                System.out.println("wrapped: " + cause.getClass().getSimpleName() +
            }
        }
    }
}
```

EXERCISE 2

Wrap the checked one

Write `static String readConfig(String name) throws java.io.IOException` that throws `new IOException("missing: " + name)` when the name starts with `missing-` and otherwise returns `name + "=ok"`. Then write `static String load(String name)` with **no** `throws` clause that calls `readConfig` and wraps any `IOException` in an `UncheckedIOException`. Read `n` names; for each call `load` and print the value, or catch the `UncheckedIOException` and print `unchecked: <cause message>`.

Example: `2` then `app missing-db` →

```
app=ok
unchecked: missing: missing-db
```

INPUT 1

```
2
app missing-db
```

OUTPUT 1

```
app=ok
unchecked: missing: missing-db
```

INPUT 2

```
1
missing-
```

OUTPUT 2

```
unchecked: missing: missing-
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

public class Main {
    static String readConfig(String name) throws IOException {
        if (name.startsWith("missing-")) throw new IOException("missing: " + name);
        return name + "=ok";
    }
    static String load(String name) {
        try {
            return readConfig(name);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String name = in.next();
            try {
                System.out.println(load(name));
            } catch (UncheckedIOException e) {
                System.out.println("unchecked: " + e.getCause().getMessage());
            }
        }
    }
}
```

QUIZ

1. A method calls `Files.readString(path)` and neither catches `IOException` nor declares it. Result?

A. Runs, throwing at run time if the file is missing

B. Compile error: unreported exception

C. A warning

D. The exception is converted to unchecked

2. Wrapping a checked exception in an unchecked one should…

A. Drop the original to save memory

B. Pass the original as the cause so its trace survives

C. Change the message to "error"

D. Only be done in `main`

3. Which is a recommended use of a **checked** exception?

A. A null argument

B. An index out of range

C. A recoverable, expected condition at an API boundary, such as a missing file

D. A violated invariant

4. Why do checked exceptions not mix well with lambdas?

A. Lambdas cannot contain `try`

B. Functional interfaces like `Function` declare no checked exceptions, so the body cannot let one escape

C. Lambdas are compiled differently

D. They do mix well

5. `public void load() throws Exception` in a library API is…

A. Fine

B. Poor: callers learn nothing and must catch everything

C. Required for I/O

D. Faster

ANSWER KEY · CHECKED VERSUS UNCHECKED — THE RULES AND THE JUDGEMENT

1. B Checked exceptions must be caught or declared — the compiler enforces it at every call site.

2. B `new UncheckedIOException(msg, e)` — the `Caused by` section is the real diagnosis.

3. C Programming errors are unchecked; expected, recoverable conditions may be checked so callers must acknowledge them.

4. B The lambda may throw only what the SAM declares; wrap inside the body or use a helper.

5. B Declare the specific checked types; if there are many unrelated ones, reconsider the method's design.

LESSON 3 · 14 MIN

try, catch and finally — the exact semantics

`try` / `catch` / `finally` looks simple and has several precise rules that interviewers probe: the order handlers are tried, what `finally` does to a `return`, what happens when both `catch` and `finally` throw, and how multi-catch and rethrow interact with the type system. This lesson pins down each rule with the smallest example that demonstrates it.

The shape

```
try {
    risky();                          // the guarded block
} catch (FileNotFoundException e) {   // handlers, tried in order; first match
    …
} catch (IOException e) {             // a broader type AFTER the narrower one
    …
} finally {
    cleanup();                        // runs whether or not an exception occurre
}
```

At least one `catch` or a `finally` is required. When an exception is thrown inside the `try`, execution jumps to the first `catch` whose type is the exception's class or a superclass; the rest of the `try` block is skipped. If no `catch` matches, `finally` runs and the exception continues up the stack.

Handler order

Handlers are checked top to bottom, and the first match wins, so a **broad type above a narrow one makes the narrow one unreachable** — and the compiler says so: "exception FileNotFoundException has already been caught". Order from most specific to most general.

Multi-catch

```
try {
    parseAndStore(input);
} catch (NumberFormatException | DateTimeParseException e) {    // one handler
    log("bad input: " + e.getMessage());
}
```

The alternatives must not be subclasses of each other (`IOException | FileNotFoundException` is an error), and the parameter `e` is implicitly `final` — its static type is the nearest common supertype. Use multi-catch when the handling is identical; separate handlers when it differs.

`finally`: always, with three exceptions

`finally` runs when the `try` block completes normally, when it throws, when a `catch` throws, and when either block executes `return`, `break` or `continue`. It does **not** run if the JVM exits (`System.exit`), the thread is killed, or the process crashes. Its purpose is cleanup that must happen regardless: closing, unlocking, restoring state. (For closing `AutoCloseable` resources, try-with-resources is better — next lesson.)

`finally` and `return`

```
static int f() {
    try {
        return 1;
    } finally {
        System.out.println("finally");     // prints, then 1 is returned
    }
}

static int g() {
    try {
        return 1;
    } finally {
        return 2;                          // OVERRIDES: g() returns 2, and ar
    }
}

static int h() {
    int x = 1;
    try {
        return x;                          // the value 1 is captured here…
    } finally {
        x = 2;                             // …so this change is not seen; h()
    }
}
```

A `return` (or `throw`) in `finally` replaces whatever the `try` / `catch` was about to do, including silently swallowing an exception — the compiler warns ("finally clause cannot complete normally") and code reviewers reject it. Never `return` from `finally`. Note also that `return x` evaluates `x` *before* `finally` runs.

When both throw

```
try {
    throw new IllegalStateException("from try");
} finally {
    throw new RuntimeException("from finally");    // this one propagates; "fr
}
```

An exception thrown in `finally` (or in a `catch`) replaces the one in flight. This is how cleanup code hides the real error — the motivation for try-with-resources' *suppressed* exceptions.

Rethrowing

```
catch (IOException e) {
    log(e);
    throw e;                                    // rethrow the same object: orig
}
catch (IOException e) {
    throw new ServiceException("load failed", e);     // wrap with cause: both
}
catch (Exception e) {
    throw e;          // Java 7+ "precise rethrow": if the try only throws IOEx
}
```

`throw e` keeps the original trace (the trace was captured at construction). Wrapping adds context and translates the type for callers; always pass the cause. Precise rethrow lets a `catch (Exception e) { … throw e; }` declare only the checked types the `try` block can actually throw, as long as `e` is not reassigned.

Scope of the `catch` parameter

`e` is local to its handler. A variable declared in the `try` block is **not** visible in `catch` or `finally` — declare it before the `try` if the handler needs it:

```
Connection c = null;
try {
    c = open();
    …
} finally {
    if (c ≠ null) c.close();
}
```

Nested `try` and the flow

A `try` inside a `catch` or `finally` is legal and sometimes needed (closing in `finally` can itself throw). Keep nesting shallow; extract a method when it grows. An exception thrown in a `catch` block is *not* caught by sibling `catch` blocks of the same `try` — only by an enclosing `try`.

`try` without `catch`

`try { … } finally { … }` is common: run cleanup, let exceptions pass through untouched. It expresses "I am not handling this, I am just tidying up" and is preferable to `catch (Exception e) { cleanup(); throw e; }`.

Cost

Entering a `try` block is free — the JVM records handler ranges in a table; there is no per-entry work. What costs is *throwing*: constructing the exception (stack walk) and unwinding. Code that never throws pays nothing for the `try`.

Interview angle

"Does `finally` run after `return`?" Yes. The return value is computed first, then `finally` runs, then the method returns.

"What if `finally` also returns?" Its return wins and any exception is discarded — never do it.

"Order of catch blocks?" Specific before general; the compiler rejects an unreachable handler.

"When does `finally` not run?" `System.exit`, JVM crash, thread death.

"Multi-catch restrictions?" Types must not be related by inheritance; the parameter is final.

Key takeaways

Handlers are tried top to bottom; first match wins; narrow before broad.

`finally` always runs (except on JVM exit) — after the return value is computed; a `return` / `throw` in `finally` overrides and hides everything.

An exception from `catch` / `finally` replaces the one in flight — the reason try-with-resources exists.

Rethrow the same object to keep the trace; wrap with a cause to add context.

`try` / `finally` without `catch` for cleanup that does not handle.

EXERCISE 1

Trace the flow

Implement `static String run(String mode)` with a `try` / `catch` / `finally` that appends to a shared `StringBuilder log`: the try block appends `try` then, by mode, `return`s `"r"` (mode `ret`), throws `IllegalStateException` (mode `throw`), or completes normally returning `"n"` (mode `ok`); the catch appends `catch` and returns `"c"`; the finally appends `finally`. Read an integer `n` and `n` modes; for each, clear the log, call `run`, and print `<log joined by ⟶⟩ ⇒ <returned value>`.

Expected:

```
ret: try⟶finally ⇒ r
throw: try⟶catch⟶finally ⇒ c
ok: try⟶finally ⇒ n
```

(the mode name is printed first, as shown)

INPUT 1

```
3
ret throw ok
```

OUTPUT 1

```
ret: try⟶finally ⇒ r
throw: try⟶catch⟶finally ⇒ c
ok: try⟶finally ⇒ n
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static List<String> log = new ArrayList<>();

    static String run(String mode) {
        try {
            log.add("try");
            if (mode.equals("ret")) return "r";
            if (mode.equals("throw")) throw new IllegalStateException("boom");
            return "n";
        } catch (IllegalStateException e) {
            log.add("catch");
            return "c";
        } finally {
            log.add("finally");
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String mode = in.next();
            log.clear();
            String result = run(mode);
            System.out.println(mode + ": " + String.join("→", log) + " ⇒ " + resul
        }
    }
}
```

EXERCISE 2

finally, counted

Write `static int classify(int x)`: inside a `try`, `return 1` when `x > 0`, throw `IllegalArgumentException` when `x < 0`, and `return 0` otherwise; the `finally` block increments a static counter `finallyRuns` **every** time. Read `n` integers; for each print `<x> → <result>` or `<x> → error`, and at the end `finallyRuns=<count>` — it equals `n`, because `finally` runs on return and on throw alike.

Example: `3` then `5 -2 0` →

```
5 → 1
-2 → error
0 → 0
finallyRuns=3
```

INPUT 1

```
3
5 -2 0
```

OUTPUT 1

```
5 → 1
-2 → error
0 → 0
finallyRuns=3
```

INPUT 2

```
1
-1
```

OUTPUT 2

```
-1 → error
finallyRuns=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static int finallyRuns = 0;

    static int classify(int x) {
        try {
            if (x > 0) return 1;
            if (x < 0) throw new IllegalArgumentException("negative");
            return 0;
        } finally {
            finallyRuns++;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            try {
                System.out.println(x + " -> " + classify(x));
            } catch (IllegalArgumentException e) {
                System.out.println(x + " -> error");
            }
        }
        System.out.println("finallyRuns=" + finallyRuns);
    }
}
```

QUIZ

1. `try { return 1; } finally { System.out.println("f"); }` prints…

A. Nothing; returns 1

B. `f`, then returns 1

C. `f`, then returns nothing

D. Compile error

2. `try { throw new A(); } finally { return 5; }` results in…

A. `A` propagates

B. The method returns 5 and the exception is silently discarded

C. Compile error

D. Both

3. `catch (IOException e) { } catch (FileNotFoundException e) { }` is…

A. Fine

B. A compile error: the second handler is unreachable

C. A warning

D. Valid only in multi-catch

4. In `catch (A | B e)`, the types must…

A. Be subclasses of one another

B. Not be related by inheritance; `e` is implicitly final

C. Both be checked

D. Both be unchecked

5. An exception thrown inside a `catch` block…

A. Is caught by the next `catch` of the same `try`

B. Propagates outward (after `finally`), replacing the original

C. Is ignored

D. Restarts the `try`

ANSWER KEY · TRY, CATCH AND FINALLY — THE EXACT SEMANTICS

1. B `finally` runs after the return value is computed and before the method actually returns.

2. B A `return` in `finally` overrides everything, including a pending exception — never write it.

3. B Handlers are tried in order and `IOException` already catches its subclass. Put narrow before broad.

4. B Alternatives in a multi-catch cannot be in a subclass relationship.

5. B Sibling handlers do not catch each other's exceptions; only an enclosing `try` can.

LESSON 4 · 12 MIN

try-with-resources and AutoCloseable

Files, sockets, database connections and locks must be released whether or not the code using them throws. Before Java 7 that meant `finally` blocks with null checks and nested `try`s that were wrong more often than right — and when both the work and the cleanup threw, the interesting exception was lost. **try-with-resources** makes the correct pattern the short one: declare the resource in the `try` header and the compiler generates the close, in the right order, preserving every exception.

The form

```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
}                                                  // br.close() is called here,
```

Any object whose type implements `AutoCloseable` (`void close() throws Exception`) — or its subinterface `Closeable` (`close() throws IOException`) — can be a resource. Streams, readers, writers, channels, `Scanner`, JDBC connections/statements/result sets, `ExecutorService` (Java 19+), `java.util.stream.Stream` (rarely needed) all qualify.

The resource variable is implicitly `final` and scoped to the `try` block. Since Java 9 an existing effectively-final variable can be used directly: `try (br) { … }`.

Several resources

```
try (InputStream in = Files.newInputStream(src);
     OutputStream out = Files.newOutputStream(dst)) {
    in.transferTo(out);
}                                                  // closed in REVERSE order: c
```

Resources are separated by semicolons and closed in reverse declaration order — later resources may depend on earlier ones, so they are torn down first. If creating the second resource throws, the first is still closed.

What it expands to

Roughly:

```
BufferedReader br = Files.newBufferedReader(path);
Throwable primary = null;
try {
    return br.readLine();
} catch (Throwable t) {
    primary = t;
    throw t;
} finally {
    if (br ≠ null) {
        if (primary ≠ null) {
            try { br.close(); } catch (Throwable suppressed) { primary.addSupp
        } else {
            br.close();
        }
    }
}
```

The two things this gets right that hand-written code got wrong: the resource is closed even if the body throws, and if **both** the body and `close()` throw, the body's exception propagates and the close exception is attached to it as a **suppressed** exception instead of replacing it.

Suppressed exceptions

```
try {
    …
} catch (IOException e) {
    for (Throwable s : e.getSuppressed()) log("also failed during close: " + s
}
```

`Throwable.getSuppressed()` returns the close failures; `printStackTrace` prints them under `Suppressed:`. Nothing is lost. In the plain `finally` version the close exception would have won and the real cause vanished.

`catch` and `finally` with resources

```
try (Connection c = pool.get()) {
    c.execute(sql);
} catch (SQLException e) {          // runs AFTER the resource is closed
    log(e);
} finally {
    metrics.record();               // runs after close and after catch
}
```

The resource is closed **before** any `catch` or `finally` of the same statement runs. So a `catch` block cannot use the resource — it is already closed — and cleanup that must see the resource open belongs in the body.

Writing an `AutoCloseable`

```
public final class Timer implements AutoCloseable {
    private final long start = System.nanoTime();
    private final String label;
    public Timer(String label) { this.label = label; }
    @Override public void close() {                  // narrow the throws clau
        System.out.println(label + ": " + (System.nanoTime() - start) / 1_000_
    }
}

try (Timer t = new Timer("load")) {
    load();
}                                                    // prints the elapsed tir
```

Any class holding something that needs releasing — a file handle, a lock, a temporary directory, a scoped setting — should implement `AutoCloseable`. Guidelines: make `close()` **idempotent** (a second call does nothing), declare the narrowest exception (`void close()` with no `throws` if it cannot fail), and do not throw from `close()` unless the failure matters. `Closeable` is for I/O; `AutoCloseable` for everything else.

Locks and scoped state

The pattern is not only for I/O. A lock guard:

```
class Guard implements AutoCloseable {
    private final Lock lock;
    Guard(Lock l) { lock = l; lock.lock(); }
    public void close() { lock.unlock(); }
}
try (Guard g = new Guard(lock)) { … }
```

Or setting and restoring a thread-local, a logging context, a temporary system property — anything with a paired undo.

What not to do

Do not close a resource inside the body *and* let try-with-resources close it — double close is at best wasted, at worst an error; idempotent `close()` makes it harmless.

Do not use try-with-resources for objects that must outlive the block (a connection stored in a field): the block closes it.

Do not implement `AutoCloseable` on classes that have nothing to release just to allow the syntax.

Do not forget it: a reader created without try-with-resources and closed in `finally` is legal and inferior. Static analysers flag "resource leak" for a reason.

Interview angle

"What does try-with-resources require of the resource?" `AutoCloseable` (or `Closeable`).

"In what order are multiple resources closed?" Reverse declaration order.

"What happens if both the body and `close()` throw?" The body's exception propagates; the close exception is suppressed and attached to it.

"Does `catch` run before or after close?" After.

"Why is it better than `finally`?" Correct in every path, shorter, and it preserves the primary exception.

Key takeaways

`try (R r = …) { … }` closes `r` on every exit path; resources are final and closed in reverse order.

Body exception wins; close exceptions are *suppressed* and retrievable — nothing is lost.

`catch` / `finally` run after the close.

Implement `AutoCloseable` for anything with a paired release — I/O, locks, scoped state — with an idempotent `close()`.

EXERCISE 1

Close order and suppressed exceptions

Write `class Res implements AutoCloseable` with a name and a flag `failOnClose`; its `close()` appends `close <name>` to a shared log and throws `IllegalStateException("close " + name)` if the flag is set. Read a line of resource specs `a b! c` (a trailing `!` means fail on close) and a word `throw` or `ok`. Open all resources in one try-with-resources (in the given order); in the body, append `body` and throw `RuntimeException("body")` if the word is `throw`. Catch the exception outside, then print the log joined by `→` and, if an exception was caught, `caught=<message> suppressed=<messages of getSuppressed(), comma-joined>`.

Example: `a b! c` with `throw` →

```
body→close c→close b→close a
caught=body suppressed=close b
```

INPUT 1

```
a b! c
throw
```

OUTPUT 1

```
body→close c→close b→close a
caught=body suppressed=close b
```

INPUT 2

```
a b c
ok
```

OUTPUT 2

```
body→close c→close b→close a
```

REFERENCE SOLUTION

```
import java.util.*;

class Res implements AutoCloseable {
    static List<String> log = new ArrayList<>();
    private final String name;
    private final boolean failOnClose;
    Res(String name, boolean failOnClose) { this.name = name; this.failOnClose = fai
    @Override public void close() {
        log.add("close " + name);
        if (failOnClose) throw new IllegalStateException("close " + name);
    }
}

public class Main {
    static Res make(String spec) {
        boolean fail = spec.endsWith("!");
        return new Res(fail ? spec.substring(0, spec.length() - 1) : spec, fail);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String[] specs = in.nextLine().trim().split("\\s+");
        String mode = in.next();
        Exception caught = null;
        try (Res a = make(specs[0]); Res b = make(specs[1]); Res c = make(specs[2]))
            Res.log.add("body");
            if (mode.equals("throw")) throw new RuntimeException("body");
        } catch (Exception e) {
            caught = e;
        }
        System.out.println(String.join("→", Res.log));
        if (caught ≠ null) {
            List<String> sup = new ArrayList<>();
            for (Throwable t : caught.getSuppressed()) sup.add(t.getMessage());
            System.out.println("caught=" + caught.getMessage() + " suppressed=" + St
        }
    }
}
```

EXERCISE 2

A resource pool that closes in order

Write `class Conn implements AutoCloseable` whose constructor prints `open <name>` and whose `close()` prints `close <name>`. Read `n` names. Open all of them in **one** try-with-resources statement is impossible for a variable count — so open them in a loop nested with recursion: `static`

`void useAll(List<String> names, int i)` opens `names.get(i)` in a try-with-resources and recurses for `i + 1`; when `i == names.size()` print `working with <n> connections`. The output shows the opens, the work, then the closes in **reverse** order.

Example: `2` then `db cache` →

```
open db
open cache
working with 2 connections
close cache
close db
```

INPUT 1

```
2
db cache
```

OUTPUT 1

```
open db
open cache
working with 2 connections
close cache
close db
```

INPUT 2

```
0
```

OUTPUT 2

```
working with 0 connections
```

REFERENCE SOLUTION

```java
import java.util.*;

class Conn implements AutoCloseable {
    final String name;
    Conn(String name) { this.name = name; System.out.println("open " + name); }
    @Override public void close() { System.out.println("close " + name); }
}

public class Main {
    static void useAll(List<String> names, int i) {
        if (i == names.size()) {
            System.out.println("working with " + names.size() + " connections");
            return;
        }
        try (Conn c = new Conn(names.get(i))) {
            useAll(names, i + 1);
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> names = new ArrayList<>();
        for (int i = 0; i < n; i++) names.add(in.next());
        useAll(names, 0);
    }
}
```

QUIZ

1. A resource in a try-with-resources header must implement…

A. `Closeable` only

B. `AutoCloseable` (or its subinterface `Closeable`)

C. `Serializable`

D. `Runnable`

2. With `try (A a = …; B b = …)`, the close order is…

A. a then b

B. b then a — reverse declaration order

C. Unspecified

D. Simultaneous

3. If the body throws X and `close()` throws Y…

A. Y propagates, X is lost

B. X propagates with Y attached as a suppressed exception

C. Both propagate

D. Neither propagates

4. A `catch` block attached to a try-with-resources runs…

A. Before the resource is closed

B. After the resource is closed

C. Only if `close()` throws

D. Instead of closing

5. A well-behaved `close()` should be…

A. Slow

B. Idempotent — a second call does nothing

C. Static

D. Private

ANSWER KEY · TRY-WITH-RESOURCES AND AUTOCLOSEABLE

1. B `AutoCloseable.close()` is what the generated code calls.

2. B Later resources may depend on earlier ones, so they are released first.

3. B The primary exception wins; `getSuppressed()` returns Y. A hand-written `finally` would have lost X.

4. B The generated close is part of the try; catch and finally come after it.

5. B Double closes happen; the JDK's streams and the try-with-resources expansion assume it is harmless.

LESSON 5 · 12 MIN

Custom exceptions and exception chaining

The JDK's exceptions describe *mechanical* failures — a null, an index, a missing file. Your program has *domain* failures: an insufficient balance, an unknown customer, an order that cannot be cancelled after shipping. Custom exception classes let callers catch those by name, carry the data needed to handle or report them, and hide the mechanical causes underneath. This lesson is how to design them, how to chain causes, and how to decide when one is warranted.

Declaring one

```
public class InsufficientFundsException extends RuntimeException {
    private final long requestedCents;
    private final long availableCents;

    public InsufficientFundsException(long requestedCents, long availableCents
        super("requested " + requestedCents + " but only " + availableCents +
        this.requestedCents = requestedCents;
        this.availableCents = availableCents;
    }

    public long requestedCents() { return requestedCents; }
    public long availableCents() { return availableCents; }
}
```

Choices made here:

Superclass: `RuntimeException` for unchecked, `Exception` for checked (previous lesson's judgement). Never `Throwable` or `Error`.

Name ends in `Exception` and states the condition, not the location.

Message built in the constructor from the facts, so every thrower produces the same, informative text.

Fields for the data a handler might use — amounts, ids, the offending value — with accessors. Exceptions are objects; use that.

Fields `final`; the class need not be, but usually should be (subclassing exceptions for a hierarchy is fine; subclassing to alter behaviour is odd).

Provide the standard constructors when the exception is general-purpose: `(String message)`, `(String message, Throwable cause)`, and if useful `(Throwable cause)`. A domain exception with specific fields, as above, often needs only its own.

Chaining: keep the cause

```
public Order load(long id) {
    try {
        return repository.find(id);
    } catch (SQLException e) {
        throw new OrderLoadException("could not load order " + id, e);      //
    }
}
```

Wrapping translates a low-level exception into one meaningful at this layer — callers of `load` should not know about SQL — while the original travels along as the **cause**. The trace then shows both:

```
OrderLoadException: could not load order 42
    at OrderService.load(OrderService.java:31)
    …
Caused by: java.sql.SQLException: connection reset
    at …
```

Rules: **always** pass the cause when wrapping (`new X(msg, e)`, or `initCause(e)` for constructors that lack the parameter); never wrap just to change the type without adding context; never catch, log and rethrow a *different* exception without the cause — that is how "connection reset" becomes an unexplained "load failed" in production logs.

Exception hierarchies

```
public class BankException extends RuntimeException { … }
public class InsufficientFundsException extends BankException { … }
public class AccountFrozenException extends BankException { … }
```

A small hierarchy lets callers choose their granularity: `catch (InsufficientFundsException e)` for the specific case, `catch (BankException e)` for "anything the bank module signals".

Keep it shallow — one base per module or layer, a handful of leaves. A hierarchy that mirrors every method is noise.

Translating across layers

Each layer of an application should throw exceptions in its own vocabulary: persistence throws data-access exceptions, services throw domain exceptions, the web layer turns them into HTTP responses. The translation points are the `catch`-and-wrap blocks at layer boundaries, and the causes preserve the full story for the logs. Letting `SQLException` reach a controller — or an `HttpStatus` reach a repository — is a design leak.

When a custom exception is warranted

Create one when at least one of these holds:

- Callers will **catch it specifically** and do something different from what they do for other failures.
- It carries **structured data** a handler needs (an id, an amount, a set of validation errors).
- It marks a **layer boundary** and hides implementation types.
- It names a **domain condition** that appears in requirements ("insufficient funds", "duplicate username").

Otherwise use the JDK's: `IllegalArgumentException` for bad arguments, `IllegalStateException` for bad timing, `UnsupportedOperationException` for unimplemented operations, `NoSuchElementException` for lookups that fail. A custom `InvalidArgumentException` that is `IllegalArgumentException` with a new name helps nobody.

Messages

A message is for the person reading the log, not for the program: state what was attempted and the values involved (`"order 42 for customer 7 not found"`), never just `"error"` or `"invalid"`. Do not include secrets or personal data. Do not depend on messages programmatically — that is what types and fields are for.

Serialization note

Exceptions are `Serializable`; if one is ever serialised (RMI, some frameworks), its fields must be serialisable too and a `serialVersionUID` avoids warnings. For ordinary applications, ignore this.

Interview angle

"How do you write a custom exception?" Extend `RuntimeException` (or `Exception`), constructor with message and cause, fields for relevant data.

"What is exception chaining?" Wrapping a caught exception as the `cause` of a new one so the original trace is preserved and the type fits the layer.

"When should you create one rather than use `IllegalArgumentException`?" When callers catch it specifically, it carries data, or it hides a layer's internals.

"Why is losing the cause bad?" The real failure and its stack trace disappear from the logs.

Key takeaways

Extend `RuntimeException` (usually) or `Exception`; name the condition; build the message from the facts; carry data in final fields.

Wrap lower-level exceptions at layer boundaries, always with the cause.

Shallow hierarchies with one base per module let callers pick their granularity.

Prefer JDK exceptions unless callers catch yours specifically or it carries data.

EXERCISE 1

Validation with data

Write `class ValidationException extends RuntimeException` carrying the field name and the rejected value, with the message `<field>: <reason> (was '<value>')`. Write `static int parseAge(String s)` that throws it for a non-integer (`"not a number"`, wrapping the `NumberFormatException` as the cause) or an age outside 0–150 (`"out of range"`).

Read an integer `n` and `n` tokens. For each, print `ok <age>` or the exception's message followed by `cause=<cause simple class name or none>`.

Example: `3` then `30 abc 200` →

```
ok 30
age: not a number (was 'abc') cause=NumberFormatException
age: out of range (was '200') cause=none
```

INPUT 1

```
3
30 abc 200
```

OUTPUT 1

```
ok 30
age: not a number (was 'abc')
cause=NumberFormatException
age: out of range (was '200') cause=none
```

INPUT 2

```
2
-1 0
```

OUTPUT 2

```
age: out of range (was '-1') cause=none
ok 0
```

REFERENCE SOLUTION

```
import java.util.*;

class ValidationException extends RuntimeException {
    private final String field;
    private final String value;

    ValidationException(String field, String value, String reason) {
        this(field, value, reason, null);
    }

    ValidationException(String field, String value, String reason, Throwable cause)
        super(field + ": " + reason + " (was '" + value + "')", cause);
        this.field = field;
        this.value = value;
    }

    String field() { return field; }
    String value() { return value; }
}

public class Main {
    static int parseAge(String s) {
        int age;
        try {
            age = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ValidationException("age", s, "not a number", e);
        }
        if (age < 0 || age > 150) throw new ValidationException("age", s, "out of ra
        return age;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String token = in.next();
            try {
                System.out.println("ok " + parseAge(token));
            } catch (ValidationException e) {
                Throwable c = e.getCause();
                System.out.println(e.getMessage() + " cause=" + (c == null ? "none"
            }
        }
    }
}
```

EXERCISE 2

Insufficient funds, with data

Write `class InsufficientFundsException extends Exception` carrying `long shortfall` (a field with a getter) and the message `short by <shortfall>`. `class Account` has a balance and `void withdraw(long amount) throws InsufficientFundsException`. Read an opening balance and `n` withdrawals; for each print `withdrew <amount>, balance <b>` or `refused: <message> (shortfall=<n>)` using the getter, and finally `refused=<count>`.

Example: `100` then `3` then `30 90 20` →

```
withdrew 30, balance 70
refused: short by 20 (shortfall=20)
withdrew 20, balance 50
refused=1
```

INPUT 1

```
100
3
30 90 20
```

OUTPUT 1

```
withdrew 30, balance 70
refused: short by 20 (shortfall=20)
withdrew 20, balance 50
refused=1
```

INPUT 2

```
0
1
1
```

OUTPUT 2

```
refused: short by 1 (shortfall=1)
refused=1
```

REFERENCE SOLUTION

```
import java.util.*;

class InsufficientFundsException extends Exception {
    private final long shortfall;
    InsufficientFundsException(long shortfall) {
        super("short by " + shortfall);
        this.shortfall = shortfall;
    }
    long getShortfall() { return shortfall; }
}

class Account {
    private long balance;
    Account(long balance) { this.balance = balance; }
    long balance() { return balance; }
    void withdraw(long amount) throws InsufficientFundsException {
        if (amount > balance) throw new InsufficientFundsException(amount - balance)
        balance -= amount;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Account acct = new Account(in.nextLong());
        int n = in.nextInt();
        int refused = 0;
        for (int i = 0; i < n; i++) {
            long amount = in.nextLong();
            try {
                acct.withdraw(amount);
                System.out.println("withdrew " + amount + ", balance " + acct.balanc
            } catch (InsufficientFundsException e) {
                refused++;
                System.out.println("refused: " + e.getMessage() + " (shortfall=" + e
            }
        }
        System.out.println("refused=" + refused);
    }
}
```

QUIZ

1. A custom exception class should usually extend…

A. `Throwable`

B. `Error`

C. `RuntimeException` (or `Exception` when checked is intended)

D. `Object`

2. Exception chaining means…

A. Throwing two exceptions

B. Passing the caught exception as the `cause` of the new one

C. Catching in a loop

D. Using `finally`

3. Fields on a custom exception are useful for…

A. Nothing — messages suffice

B. Giving handlers structured data (ids, amounts, the offending value)

C. Serialization only

D. Performance

4. Creating `InvalidArgumentException extends IllegalArgumentException` with no extra behaviour is…

A. Best practice

B. Usually pointless — prefer the JDK exception unless callers catch yours specifically or it carries data

C. Required by frameworks

D. Faster

5. Each application layer should…

A. Throw `Exception`

B. Throw exceptions in its own vocabulary, wrapping lower-level ones with causes at the boundary

C. Let `SQLException` reach the UI

D. Catch everything

ANSWER KEY · CUSTOM EXCEPTIONS AND EXCEPTION CHAINING

1. C Never `Throwable` or `Error`; choose checked or unchecked deliberately.

2. B The `Caused by` section of the trace is the original failure; losing it loses the diagnosis.

3. B Exceptions are objects; a handler can read `e.requestedCents()` instead of parsing the message.

4. B A new name with no new information adds a type without adding value.

5. B Translation at boundaries keeps implementation details from leaking upward while preserving the evidence.

LESSON 6 · 13 MIN

Exception handling that survives production

The syntax of exceptions is a day's work; the judgement takes longer. Most exception bugs in real systems are not syntax — they are swallowed failures, exceptions used as `if` statements, `catch (Exception e)` hiding a `NullPointerException`, and logs that say "error" with no context. This lesson is the set of practices that experienced Java teams enforce in review, each with the failure it prevents.

1. Never swallow

```
try { save(order); } catch (Exception e) { }                    // the order w
try { save(order); } catch (Exception e) { e.printStackTrace(); }   // stderr,
```

An empty `catch` turns a failure into silent data loss. If you genuinely intend to ignore an exception (closing a stream you no longer care about), say so with a comment and catch the narrowest type. Otherwise handle it, or let it propagate.

2. Catch the narrowest type you can handle

`catch (Exception e)` catches `NullPointerException`, `ClassCastException` and every other bug along with the `IOException` you meant. Bugs then look like transient failures and get retried or ignored. Catch `IOException`; let the bugs crash loudly so they get fixed. `catch (Throwable t)` catches `OutOfMemoryError` too — reserve it for a thread's top-level loop that must not die, and rethrow `Error`s even there.

3. Log or rethrow — not both

```
catch (IOException e) {
    log.error("load failed", e);
    throw new LoadException("load failed", e);      // now the same failure is
}
```

Each layer that logs and rethrows produces another copy of the trace; a single failure becomes five pages. Decide where the exception is *handled* and log there, once, with the cause. Everywhere else, wrap (with cause) or let it pass.

---

4. Do not use exceptions for control flow

```
try { return Integer.parseInt(s); } catch (NumberFormatException e) { return
try { return array[i]; } catch (ArrayIndexOutOfBoundsException e) { return nu
```

Throwing is expensive (a stack walk per construction) and, more importantly, an exception in normal flow hides the *real* exceptions of that type. Test conditions you can test; use exceptions for the exceptional. When an API gives you only a throwing form (`parseInt`), catching the specific exception is fine.

---

5. Fail fast, at the boundary

Validate arguments at the top of public methods and constructors and throw `IllegalArgumentException` / `NullPointerException` immediately. A bad value that travels through six calls before exploding is far harder to diagnose than one rejected at the door. `Objects.requireNonNull(x, "x")` is the one-liner.

---

6. Preserve the cause and add context

`throw new ServiceException("saving order " + id + " for customer " + customerId, e)` — the message says what was being attempted with which data; the cause says why it failed. A wrapped exception without a cause deletes the only useful stack trace; a message without context makes the trace the only clue.

---

7. Keep `try` blocks small

A `try` around fifty lines with one `catch (Exception e)` at the end tells nobody which of the fifty lines failed or which failures were anticipated. Wrap the one call that can throw; handle that; move on. If several calls throw different things, that is several small `try`s or a method extraction.

8. Clean up with try-with-resources

Anything closeable goes in a try-with-resources header (previous lesson). `finally` is for cleanup of things that are not `AutoCloseable` — and even then, a small `AutoCloseable` wrapper is often cleaner.

9. Do not return `null` or a sentinel to signal failure when an exception fits — and vice versa

Absent-but-normal (a lookup that may miss) → `Optional` or a documented `null`/empty collection. Failure of an operation that should have worked → exception. A method that returns `null` on error forces every caller to remember a check; a method that throws for a normal miss forces every caller into a `try`. Match the mechanism to how common and how expected the outcome is.

10. `finally` never returns or throws

Covered in lesson 3: a `return` or `throw` in `finally` discards the in-flight exception. Reviewers reject it on sight.

Assertions

```
assert index ⩾ 0 : "negative index " + index;
```

`assert` checks an *internal* invariant and throws `AssertionError` when it fails — **only if the JVM runs with** `-ea`. By default assertions are disabled, so they must never validate user input or arguments (use `if` + `IllegalArgumentException`), and code must not depend on their side effects. They are documentation that runs in tests and development.

Interrupts

`InterruptedException` is special: catching it and ignoring it breaks the thread's cancellation. Either propagate it (`throws InterruptedException`) or restore the flag:

`Thread.currentThread().interrupt();` before wrapping. Module 17.

A handling checklist

Question	Answer
Can this code *fix* the problem here?	Handle it: retry, fall back, default. Log at most once.
Can it add context?	Wrap with a cause and a message naming the attempt; rethrow.
Neither?	Do not catch. Let it propagate to a layer that can.
Is it a bug (NPE, IOOBE, ISE)?	Do not catch it; fix the code.
Is it cleanup?	try-with-resources or `try` / `finally`, no `catch`.

Where exceptions are finally handled

Every application has a few *top-level* handlers: the request loop of a server that converts exceptions to error responses, the `main` of a CLI that prints a message and exit code, a thread pool's uncaught-exception handler that logs. Those are the places for `catch (Exception e)`, full logging, and user-facing translation. Everything below them should mostly wrap or ignore, so the trace arrives intact.

Interview angle

"What is wrong with an empty catch block?" It hides failures; the program continues in a broken state with no record.

"Why not `catch (Exception e)` everywhere?" It catches programming errors and treats them as recoverable, hiding bugs.

"Log and rethrow?" Duplicates logs; log where handled, once.

"Are assertions enabled by default?" No — `-ea`; never use them for argument validation.

"Exceptions vs return codes?" Exceptions cannot be ignored silently, carry a trace and separate error handling from logic; return codes are for expected outcomes.

Key takeaways

Never swallow; catch the narrowest type; log once where handled, otherwise wrap with cause or let it propagate.

Exceptions are for the exceptional — test what you can test; fail fast at the boundary with informative messages.

Small `try` blocks; try-with-resources for cleanup; nothing returns or throws from `finally`.

`assert` is off by default: internal invariants only. Restore the interrupt flag when catching `InterruptedException`.

EXERCISE 1

Fix the handling

The program below swallows failures and uses exceptions for control flow. Rewrite `sumValid` so that: it validates each token with a **check** (`matches("-?\\d+")`) rather than catching `NumberFormatException` ; it counts invalid tokens instead of ignoring them; and any token whose value is negative causes an `IllegalArgumentException` with the message `negative: <token>` that propagates to `main` , where it is caught once and printed as `error: <message>` .

Read an integer `n` and `n` tokens; print `sum=<sum> invalid=<count>` or the error line.

Example: `4` then `3 x 4 y` → `sum=7 invalid=2` ; `2` then `3 -1` → `error: negative: -1`

INPUT 1

```
4
3 x 4 y
```

OUTPUT 1

```
sum=7 invalid=2
```

INPUT 2

```
2
3 -1
```

OUTPUT 2

```
error: negative: -1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static int invalid = 0;

    static long sumValid(String[] tokens) {
        long sum = 0;
        for (String t : tokens) {
            if (!t.matches("-?\\d+")) {
                invalid++;
                continue;
            }
            long v = Long.parseLong(t);
            if (v < 0) throw new IllegalArgumentException("negative: " + t);
            sum += v;
        }
        return sum;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String[] tokens = new String[n];
        for (int i = 0; i < n; i++) tokens[i] = in.next();
        try {
            long sum = sumValid(tokens);
            System.out.println("sum=" + sum + " invalid=" + invalid);
        } catch (IllegalArgumentException e) {
            System.out.println("error: " + e.getMessage());
        }
    }
}
```

EXERCISE 2

Retry with a budget

Simulate a flaky call. Read `maxAttempts` and `n` scenarios, each a string of `F` (fail) and `S` (succeed) describing what successive attempts would do. Write `static String callWithRetry(String outcomes, int maxAttempts)` that tries up to `maxAttempts` times, throwing `IllegalStateException("attempt <k> failed")` internally on `F` and catching it to retry; on `S` return `ok after <k> attempts`. If the budget is exhausted, rethrow the **last** failure with the attempts so far as suppressed exceptions and let the caller print `gave up: <message> (suppressed=<count>)`. Never swallow silently.

Example: `3` then `2` then `FS FFF` →

```
ok after 2 attempts
gave up: attempt 3 failed (suppressed=2)
```

INPUT 1

```
3
2
FS FFF
```

OUTPUT 1

```
ok after 2 attempts
gave up: attempt 3 failed (suppressed=2)
```

INPUT 2

```
1
2
S F
```

OUTPUT 2

```
ok after 1 attempts
gave up: attempt 1 failed (suppressed=0)
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static String attempt(char outcome, int k) {
        if (outcome == 'F') throw new IllegalStateException("attempt " + k + " faile
        return "ok after " + k + " attempts";
    }

    static String callWithRetry(String outcomes, int maxAttempts) {
        List<IllegalStateException> failures = new ArrayList<>();
        for (int k = 1; k ⩽ maxAttempts; k++) {
            char outcome = k - 1 < outcomes.length() ? outcomes.charAt(k - 1) : 'F';
            try {
                return attempt(outcome, k);
            } catch (IllegalStateException e) {
                failures.add(e);
            }
        }
        IllegalStateException last = failures.get(failures.size() - 1);
        for (int i = 0; i < failures.size() - 1; i++) last.addSuppressed(failures.ge
        throw last;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int maxAttempts = in.nextInt();
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String outcomes = in.next();
            try {
                System.out.println(callWithRetry(outcomes, maxAttempts));
            } catch (IllegalStateException e) {
                System.out.println("gave up: " + e.getMessage() + " (suppressed=" +
            }
        }
    }
}
```

QUIZ

1. An empty `catch (Exception e) { }` block…

A. Is harmless

B. Silently hides failures, including bugs, so the program continues in a broken state

C. Improves performance

D. Is required by the compiler

2. Logging an exception and then rethrowing it wrapped, at every layer, causes…

A. Nothing

B. Duplicate traces of one failure filling the logs

C. Lost causes

D. Compile errors

3. Using `catch (ArrayIndexOutOfBoundsException e)` instead of checking the index is…

A. Idiomatic

B. Exceptions as control flow: slow and it hides real errors — check the condition

C. Required for arrays

D. Faster

4. Assertions (`assert`) are…

A. Always on

B. Off unless the JVM runs with `-ea`, so never use them for argument validation

C. A replacement for exceptions

D. Checked at compile time

5. Catching `InterruptedException` and ignoring it…

A. Is fine

B. Breaks cancellation — propagate it or call `Thread.currentThread().interrupt()` before wrapping

C. Speeds up the thread

D. Is required

ANSWER KEY · EXCEPTION HANDLING THAT SURVIVES PRODUCTION

1. B If ignoring is intended, catch the narrowest type and say why in a comment; otherwise handle or propagate.

2. B Log once where the exception is handled; elsewhere wrap with cause or let it propagate.

3. B Test conditions you can test; reserve catching for APIs that offer no check (`parseInt`).

4. B They document internal invariants and run in tests/development; production usually runs without them.

5. B The interrupt flag is cleared when the exception is thrown; restoring it lets callers up the stack notice.

CHECKPOINT · 24 MIN

Checkpoint — Exceptions

This checkpoint covers the exception hierarchy, checked versus unchecked, `try` / `catch` / `finally` semantics, try-with-resources, custom exceptions with chaining, and the handling practices.

How it works. Fourteen questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- Which classes are checked, which unchecked, and where `Error` sits.
- What `finally` does to a pending `return`, and what a `return` inside `finally` does.
- The order in which resources are closed and what happens to a `close()` exception.
- Why you pass the cause when wrapping, and why you log once.
- Why `catch (Exception e)` around business logic is dangerous.

The programs are a validating parser that reports every failure with a custom exception carrying data, and a resource-managed processor that proves close order and suppressed exceptions.

EXERCISE 1

Config loader with layered exceptions

Write `class ConfigException extends RuntimeException` (message + cause). Write `static int parsePort(String line)` that expects `port=<number>`: for a line without `=` throw `IllegalArgumentException("malformed: " + line)`; for a non-numeric value wrap the `NumberFormatException` in a `ConfigException("bad port: " + value, e)`; for a port outside 1–65535 throw `ConfigException("port out of range: " + n)` with no cause.

Read an integer `k` and `k` lines. For each, print `port=<n>`, or `config: <message> cause=<simple class of cause or none>` for a `ConfigException`, or `arg: <message>` for an `IllegalArgumentException`. Catch the two types separately, most specific first.

Example: `4` then `port=8080`, `port=abc`, `port=70000`, `nonsense` →

```
port=8080
config: bad port: abc cause=NumberFormatException
config: port out of range: 70000 cause=none
arg: malformed: nonsense
```

INPUT 1

```
4
port=8080
port=abc
port=70000
nonsense
```

OUTPUT 1

```
port=8080
config: bad port: abc
cause=NumberFormatException
config: port out of range: 70000
cause=none
arg: malformed: nonsense
```

INPUT 2

```
2
port=1
port=0
```

OUTPUT 2

```
port=1
config: port out of range: 0 cause=none
```

REFERENCE SOLUTION

```java
import java.util.*;

class ConfigException extends RuntimeException {
    ConfigException(String message) { super(message); }
    ConfigException(String message, Throwable cause) { super(message, cause); }
}

public class Main {
    static int parsePort(String line) {
        int eq = line.indexOf('=');
        if (eq == -1) throw new IllegalArgumentException("malformed: " + line);
        String value = line.substring(eq + 1);
        int n;
        try {
            n = Integer.parseInt(value);
        } catch (NumberFormatException e) {
            throw new ConfigException("bad port: " + value, e);
        }
        if (n < 1 || n > 65535) throw new ConfigException("port out of range: " + n)
        return n;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < k; i++) {
            String line = in.nextLine().trim();
            try {
                System.out.println("port=" + parsePort(line));
            } catch (ConfigException e) {
                Throwable c = e.getCause();
                System.out.println("config: " + e.getMessage() + " cause=" + (c == r
            } catch (IllegalArgumentException e) {
                System.out.println("arg: " + e.getMessage());
            }
        }
    }
}
```

EXERCISE 2

Transaction with guaranteed rollback

Write `class Tx implements AutoCloseable` with a boolean `committed`; `commit()` sets it, and `close()` appends `commit` or `rollback` to a shared log depending on the flag. Read an integer `n` and `n` lines of commands: `ok` (body commits), `fail` (body throws

`IllegalStateException("failed")` before committing), `skip` (body neither throws nor commits). Run each in `try (Tx tx = new Tx()) { … }`, catch the exception outside, and print `<command>: <log entry> <caught message or ->` per line.

Expected for `ok fail skip`:

```
ok: commit -
fail: rollback failed
skip: rollback -
```

INPUT 1

```
3
ok fail skip
```

OUTPUT 1

```
ok: commit -
fail: rollback failed
skip: rollback -
```

REFERENCE SOLUTION

```java
import java.util.*;

class Tx implements AutoCloseable {
    static List<String> log = new ArrayList<>();
    private boolean committed = false;
    void commit() { committed = true; }
    @Override public void close() {
        log.add(committed ? "commit" : "rollback");
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            Tx.log.clear();
            String caught = "-";
            try (Tx tx = new Tx()) {
                if (cmd.equals("fail")) throw new IllegalStateException("failed");
                if (cmd.equals("ok")) tx.commit();
            } catch (IllegalStateException e) {
                caught = e.getMessage();
            }
            System.out.println(cmd + ": " + Tx.log.get(0) + " " + caught);
        }
    }
}
```

EXERCISE 3

A calculator that never crashes

Read expression lines `<a> <op> <b>` until end of input and evaluate each, turning every failure into a message: a non-integer operand → `error: bad operand '<token>'` (`NumberFormatException`); division or modulo by zero → `error: division by zero` (`ArithmeticException`); an unknown operator → `error: unknown operator '<op>'` (throw and catch your own `UnknownOperatorException extends Exception`); a line with fewer than three tokens → `error: malformed line` . Print results as `<a> <op> <b> = <value>` . Finish with `ok=<count> errors=<count>` .

Example input

```
6 * 7
1 / 0
2 ^ 3
x + 1
```

→

```
6 * 7 = 42
error: division by zero
error: unknown operator '^'
error: bad operand 'x'
ok=1 errors=3
```

INPUT 1

```
6 * 7
1 / 0
2 ^ 3
x + 1
```

OUTPUT 1

```
6 * 7 = 42
error: division by zero
error: unknown operator '^'
error: bad operand 'x'
ok=1 errors=3
```

INPUT 2

```
5 %
10 % 3
```

OUTPUT 2

```
error: malformed line
10 % 3 = 1
ok=1 errors=1
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

class UnknownOperatorException extends Exception {
    UnknownOperatorException(String op) { super(op); }
}

public class Main {
    static long apply(long a, String op, long b) throws UnknownOperatorException {
        switch (op) {
            case "+": return a + b;
            case "-": return a - b;
            case "*": return a * b;
            case "/": return a / b;
            case "%": return a % b;
            default: throw new UnknownOperatorException(op);
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int ok = 0, errors = 0;
        while ((line = in.readLine()) != null) {
            if (line.isBlank()) continue;
            String[] t = line.trim().split("\\s+");
            String bad = null;
            try {
                if (t.length < 3) { System.out.println("error: malformed line"); err
                bad = t[0];
                long a = Long.parseLong(t[0]);
                bad = t[2];
                long b = Long.parseLong(t[2]);
                System.out.println(a + " " + t[1] + " " + b + " = " + apply(a, t[1],
                ok++;
            } catch (NumberFormatException e) {
                System.out.println("error: bad operand '" + bad + "'");
                errors++;
            } catch (ArithmeticException e) {
                System.out.println("error: division by zero");
                errors++;
            } catch (UnknownOperatorException e) {
                System.out.println("error: unknown operator '" + e.getMessage() + "'
                errors++;
            }
        }
        System.out.println("ok=" + ok + " errors=" + errors);
```

```
    }
}
```

CHECKPOINT QUESTIONS

1. The root of the exception hierarchy is…

A. `Exception`

B. `Throwable`

C. `Error`

D. `RuntimeException`

2. Which pair is (checked, unchecked)?

A. (`NullPointerException`, `IOException`)

B. (`IOException`, `IllegalStateException`)

C. (`OutOfMemoryError`, `SQLException`)

D. (`ArithmeticException`, `ClassCastException`)

3. `throws` on a method lists…

A. Every exception it might throw

B. The checked exceptions it may let escape (unchecked ones optionally, for documentation)

C. Only unchecked exceptions

D. Errors

4. `int x = 1; try { return x; } finally { x = 2; }` returns…

A. 2

B. 1 — the return value was evaluated before `finally` ran

C. Compile error

D. 0

5. Rethrowing with `throw e;` versus `throw new X(e);` differs in that…

A. Nothing

B. `throw e` keeps the same object and trace; wrapping adds context and a new type while preserving the original as cause

C. Wrapping loses the trace

D. `throw e` is not allowed

6. try-with-resources on `Closeable` versus `AutoCloseable` …

A. Only `Closeable` works

B. Both work; `Closeable` is the I/O-specific subinterface whose `close()` throws `IOException`

C. Only `AutoCloseable` works

D. Neither — it needs `Resource`

7. `getSuppressed()` returns…

A. The cause

B. Exceptions thrown while closing resources after a primary exception

C. Ignored exceptions

D. Nothing useful

8. A custom exception's message should…

A. Say "error"

B. State what was attempted and the values involved, without secrets

C. Be empty

D. Contain the stack trace

9. Which is the correct handling when you cannot fix or add context?

A. Catch and log

B. Catch and ignore

C. Do not catch — let it propagate to a layer that can handle it

D. Catch and return null

10. Why are `try` blocks free when no exception is thrown?

A. They are not free

B. Handlers are stored in a table; entering a `try` does no work — only throwing costs

C. The JIT removes them

D. Exceptions are pre-allocated

11. `catch (Throwable t)` at the top of a server's request loop is…

A. Always wrong

B. Acceptable there — to log and keep the process alive — while still rethrowing `Error`s

C. Required everywhere

D. A compile error

12. Absent-but-normal results (a lookup miss) are best signalled with…

A. An exception

B. `Optional` (or a documented empty result), not an exception

C. `null` always

D. `System.exit`

13. `assert x > 0;` in production code with default JVM flags…

A. Throws `AssertionError` when false

B. Does nothing — assertions are disabled without `-ea`

C. Logs a warning

D. Is a compile error

14. After catching `InterruptedException` inside a helper, the right move is…

A. Ignore it

B. Restore the flag with `Thread.currentThread().interrupt()` (or propagate the exception)

C. Call `Thread.stop`

D. Retry the sleep

ANSWER KEY · CHECKPOINT — EXCEPTIONS

1. B `Throwable` has two direct subclasses: `Error` and `Exception`.

2. B `IOException` extends `Exception`; `IllegalStateException` extends `RuntimeException`.

3. B The compiler tracks only checked exceptions through `throws` clauses.

4. B The value is captured at the `return` statement; `finally` cannot change it (unless it returns itself).

5. B Both preserve the evidence when the cause is passed; wrapping is for translating between layers.

6. B `Closeable extends AutoCloseable`; either qualifies.

7. B try-with-resources attaches close failures to the primary exception rather than losing either.

8. B The message is for the person reading the log; the type and fields are for the program.

9. C Catching without handling only hides or duplicates; propagation preserves the evidence for the real handler.

10. B Cost lives in constructing (stack walk) and unwinding, not in guarding.

11. B Top-level handlers are the one place for the broadest catch; business code should never have it.

12. B Exceptions are for failures of operations that should have worked; expected outcomes get ordinary return values.

13. B Never rely on assertions for validation that must run.

14. B The flag is cleared on throw; restoring it keeps the cancellation signal alive for callers.