

Class

15

CSC 240 Computer Science III

Exceptions III: Propagation, Causes & try-with-resources

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 10 · Tuesday, October 27, 2026
wcu.ninja/csc240

Where we left off, and today's plan

Thursday we got through: `throw` vs `throws`, which built-in exception to throw, your own exception class, checked or unchecked, and we **started** the solution of the Pokémon `setLevel` exercise.

First 15 min: finish that solution (TODO 2–5), including the checked-or-unchecked question.

Then (Class 14's second half): how an exception travels up the call stack, wrapping it with a **cause**, try-with-resources, overriding + `throws`, six design rules, a compiles/runs/crashes quiz.

If time: start the Pokédex loader v2 (it is a small HW3). **Thursday (Class 16):** Ch. 11.3, binary files, random access, serialization (HW3 Q5).

HW3 (Warehouse) due **Tue Nov 10, 11:59 PM**. After today you can do Q1–Q4. Code: Class 13 zip (`04_...`) and Class 14 zip (`01_...–05_...`).

Solution, TODO 2–3: `setLevel` stops ignoring bad levels

Before (Classes 5–13) and TODO 2

```
// before: a bad level is SILENTLY IGNORED
public void setLevel(int level) {
    if (level >= 1 && level <= 100)
        this.level = level;
}

// TODO 2: unchecked, built-in
public void setLevel(int level) {
    if (level < 1 || level > 100)
        throw new IllegalArgumentException(
            "level must be 1-100, got " + level);
    this.level = level;
}

// compiles with NO change to Main ...
// ... and Mewtwo:150 now crashes the loop:
Pikachu -> Lv.25
Exception in thread "main"
    java.lang.IllegalArgumentException:
        level must be 1-100, got 150
// Eevee, Snorlax, Charizard: never reached
```

TODO 3: your own checked exception

```
// InvalidLevelException.java (new file)
public class InvalidLevelException
    extends Exception {           // checked
    private int level;

    public InvalidLevelException(int level) {
        super("level must be 1-100, got " + level);
        this.level = level;
    }
    public int getLevel() { return level; }
}

// Pokemon.java
public void setLevel(int level)
    throws InvalidLevelException { // needed!
    if (level < 1 || level > 100)
        throw new InvalidLevelException(level);
    this.level = level;
}

Main.java:53: error: unreported exception
    InvalidLevelException; must be caught or
    declared to be thrown
```

The compile error is the point: a checked exception forces every caller of `setLevel` to decide what to do. That leads to TODO 4.

Solution, TODO 4: catch it in the loop, count it

```
int rejected = 0;
for (String entry : entries) {
    String[] parts = entry.split(":");
    Pokemon p = find(dex, parts[0]);
    try {
        int level = Integer.parseInt(parts[1]);
        p.setLevel(level);
        System.out.println(p.getName() + " -> Lv."
                           + p.getLevel());
    } catch (InvalidLevelException e) {
        System.out.println("rejected: " + e.getMessage()
                           + " (" + p.getName() + " stays Lv."
                           + p.getLevel() + ")");
        rejected++;
    } catch (NumberFormatException e) {
        System.out.println("rejected: " + e.getMessage());
        rejected++;
    }
}

// Pikachu -> Lv.25
// rejected: level must be 1-100, got 150 (Mewtwo stays Lv.1)
// rejected: For input string: "twelve"
// Snorlax -> Lv.100
// rejected: level must be 1-100, got -3 (Charizard stays Lv.1)
// Trainer's team: 5 Pokemon, 3 entries rejected
```

The `try` is **inside** the loop: one bad entry is skipped, the next one still runs. (Today's propagation slides are about exactly this choice.)

Two catches, two different problems: a level that is not a number (`parseInt`), and a number that is not a valid level (`setLevel`).

"stays Lv.1" is true because `setLevel` checks **before** it assigns. Validate first, then store.

`p` is declared **outside** the `try`, so the catch block can still use it.

Why `e.getMessage()` and not our own text? The exception already says what went wrong and with which value.

Class 13 zip → 04_pokemon_level_starter/. Did yours print the same 6 lines?

TODO 5: checked or unchecked here?

extends `Exception` (**what we did**)

`setLevel` must say `throws InvalidLevelException`.

Every caller **must** catch it or pass it on, or the code does not compile.

Fits when the bad value comes **from outside**: a trainer typed it, a file contained it. The caller can do something useful (skip, ask again).

Cost: every call site needs a `try`, even in code where the level is a constant like `setLevel(5)`.

extends `RuntimeException`

No `throws` needed. `Main` compiles **without** any catch.

Forget the catch and `Mewtwo:150` crashes the loop again, like TODO 2.

Fits when a bad value means a **bug** in the calling code, like `IllegalArgumentException` for a negative radius in `Circle`.

This is the choice the Java library makes for most argument checks.

Both are defensible. Rule of thumb from Thursday: checked for things the world does to you, unchecked for programmer mistakes.

Warm-up: what prints?

```
static int f(int x) {
    try {
        System.out.println("A");
        if (x < 0) throw new IllegalArgumentException("neg");
        System.out.println("B");
        return 1;
    } catch (IllegalArgumentException e) {
        System.out.println("C " + e.getMessage());
        return 2;
    } finally {
        System.out.println("D");
    }
}

public static void main(String[] args) {
    System.out.println(f(5));
    System.out.println(f(-5));
    System.out.println("E");
}
```

Two minutes. Write the output on paper, in order, before you run anything.

Then: change

`IllegalArgumentException` in the **catch** (not the throw) to `NullPointerException`. What prints now? Where does the program stop?

Reminders from last week: `throw` ends the try block at once; `finally` runs before any `return` and before any propagation; an uncaught exception ends the program with a trace.

Answer on the next slide.

Warm-up: answer

```
// f(5):
A
B
D
1
// f(-5):
A
C neg
D
2
E

// catch changed to NullPointerException, f(-5):
A
D
Exception in thread "main"
  java.lang.IllegalArgumentException: neg
    at Main.f(Main.java:4)
    at Main.main(Main.java:17)
// no 2, no E: main ended too
```

`f(5)`: no exception, so `A B`, then `return 1` pauses, `finally` prints `D`, then `1` is printed by `main`.

`f(-5)`: `throw` skips `B`, the catch prints `C neg`, `return 2` pauses for `finally (D)`, then `2`.

With the wrong catch type, nothing in `f` matches. `finally` **still prints D**, then the exception reaches `main`, which has no catch either. The JVM prints the trace and stops: no `2`, no `E`.

That last case is today's topic: an exception **travels**, and where it is caught decides what survives.

TODAY

Where to Catch, and What to Tell

An exception does not have to be caught where it happens. Today: how it travels, how to hand the caller a better one, how Java can close your files for you, and a few rules for designing code that fails well.

Propagation: where you catch decides what gets skipped

A) one try around the whole call

```
static int loadTeamA(String[] team) {
    int total = 0;
    try {
        for (String entry : team) {
            total += parseLevel(entry);
            System.out.println(" added " + entry);
        }
        return total;
    } finally {
        System.out.println(" loadTeam finally,"
            + " total so far " + total);
    }
}
// main:
try {
    System.out.println("total = " + loadTeamA(team));
} catch (NumberFormatException e) {
    System.out.println("main caught: "
        + e.getMessage() + " -> no total at all");
}
```

B) one try around each entry

```
static int loadTeamB(String[] team) {
    int total = 0;
    for (String entry : team) {
        try {
            total += parseLevel(entry);
            System.out.println(" added " + entry);
        } catch (NumberFormatException e) {
            System.out.println(" skipped " + entry);
        }
    }
    return total;
}
// main:
System.out.println("total = " + loadTeamB(team));

// parseLevel, used by both:
static int parseLevel(String entry) {
    try {
        return Integer.parseInt(entry.split(":")[1].trim());
    } finally {
        System.out.println(" parseLevel finally ("
            + entry + ")");
    }
}
```

team = {"Pikachu:25", "Mewtwo:seventy", "Eevee:12"}. Predict both outputs. (Class 14 zip → 01_propagation/)

...and every finally on the way up still runs

```
A) catch in main, around the whole call:
    parseLevel finally (Pikachu:25)
    added Pikachu:25
    parseLevel finally (Mewtwo:seventy)
    loadTeam finally, total so far 25
main caught: For input string: "seventy"
    -> no total at all
```

```
B) catch inside the loop:
    parseLevel finally (Pikachu:25)
    added Pikachu:25
    parseLevel finally (Mewtwo:seventy)
    skipped Mewtwo:seventy
    parseLevel finally (Eevee:12)
    added Eevee:12
total = 37
```

A: the exception leaves `parseLevel` (its `finally` runs), leaves `loadTeamA` (its `finally` runs: total so far 25), and is caught in `main`. Eevee is never read; `return total` never happens.

B: caught right next to the bad entry. Only that entry is skipped; the loop continues.

The exception passes through every method between the `throw` and the `catch`. Each one **ends immediately**, after its `finally`.

Neither is "right". Catch **high** when one bad part spoils the whole result (a corrupt header). Catch **low** when the parts are independent (CSV lines).

This is the same decision you made in the Pokédex loader: inner try = per line, outer try = per file.

Wrap and re-throw: hand the caller a better exception

```
public class BadLineException extends Exception {
    private int lineNo;
    public BadLineException(int lineNo, String line,
                           Throwable cause) {
        super("line " + lineNo + " is bad: \"" + line
              + "\"", cause);          // (message, cause)
        this.lineNo = lineNo;
    }
    public int getLineNo() { return lineNo; }
}

static int parseLevel(int lineNo, String line)
    throws BadLineException {
    try {
        return Integer.parseInt(line.split(":")[1].trim());
    } catch (NumberFormatException
            | ArrayIndexOutOfBoundsException e) {
        throw new BadLineException(lineNo, line, e); // wrap
    }
}

// caller:
} catch (BadLineException e) {
    System.out.println(e.getMessage() + " [cause: "
        + e.getCause().getClass().getSimpleName() + "]");
}
// line 2 is bad: "Mewtwo:seventy" [cause: NumberFormatException]
// line 3 is bad: "Eevee" [cause: ArrayIndexOutOfBoundsException]
```

A `catch` block may **throw**. Catch the low-level exception, build a new one that says what it means **here**, throw that.

`super(message, cause)`: every `Throwable` can carry the original exception. `getCause()` returns it, so nothing is lost.

The caller now catches **one** type (`BadLineException`) and gets a message with the line number and the text.

This is how libraries work: a database driver throws `SQLException` with a cause, not `ArrayIndexOutOfBoundsException`.

`throws BadLineException` in the header: it is checked.

Class 14 zip → 02_cause/

Reading a chained stack trace: "Caused by:"

```
BadLineException: line 2 is bad: "Mewtwo:seventy"
  at Main.parseLevel(Main.java:10)
  at Main.main(Main.java:30)
Caused by: java.lang.NumberFormatException:
  For input string: "seventy"
  at java.base/...NumberFormatException.forInputString(...)
  at java.base/...Integer.parseInt(Integer.java:668)
  at java.base/...Integer.parseInt(Integer.java:786)
  at Main.parseLevel(Main.java:8)
  ... 1 more

// Same shape when a framework wraps your bug:
// javax.servlet.ServletException: request failed
//   at ...
// Caused by: java.lang.NullPointerException
//   at com.school.Grades.average(Grades.java:41)  <- yours
```

Top block: the exception that was **thrown last** (the wrapper) and where it was thrown from (Main.java:10, the `throw new BadLineException()`).

`Caused by:` block: the **original** exception, where it was thrown (Main.java:8, the `parseInt`).

Rule from Class 12 still applies: find the first line from **your** file. Now there may be one in each block; the deepest `Caused by:` is usually the real bug.

`... 1 more:` frames identical to the block above, not printed again.

`e.printStackTrace()` prints the whole chain. `getMessage()` prints only the wrapper's text.

Class 14 zip → 02_cause/ (second half of main)

try-with-resources: Java closes the file for you

Before Java 7: finally + close()

```
static int countOld(String file)
    throws IOException {
    BufferedReader br = null;    // outside the try
    try {
        br = new BufferedReader(
            new FileReader(file));
        int n = 0;
        while (br.readLine() != null) n++;
        return n;
    } finally {
        if (br != null) br.close();
    }
}
```

Java 7+: declare it inside try ()

```
static int countNew(String file)
    throws IOException {
    try (BufferedReader br = new BufferedReader(
        new FileReader(file))) {
        int n = 0;
        while (br.readLine() != null) n++;
        return n;
    }    // br.close() happens HERE, always
}

// several resources: separate with ;
try (Scanner in = new Scanner(new File(a));
    PrintWriter out = new PrintWriter(b)) {
    ...
}    // closed in REVERSE order: out, then in

// catch / finally may follow, as usual:
try (BufferedReader br = ...) {
    ...
} catch (FileNotFoundException e) {
    System.out.println("cannot open " + file);
}
```

Any [AutoCloseable](#) works: Scanner, readers, writers, streams. Class 14 zip → [03_try_with_resources/](#)

Chapter 10 meets Chapter 11: overriding and throws

```
interface Loader {
    int load(String file) throws IOException; // contract
}

class CsvLoader implements Loader {
    public int load(String f) throws FileNotFoundException
    { ... }    // OK: NARROWER (is-a IOException)
}

class MemoryLoader implements Loader {
    public int load(String f) { ... }
    // OK: throws NOTHING
}

class PickyLoader implements Loader {
    public int load(String f) throws IOException {
        if (f.startsWith("#"))
            throw new IllegalArgumentException("no #");
        ...    // OK: UNCHECKED is never restricted
    }
}

class BadLoader implements Loader {
    public int load(String f) throws Exception { ... }
    // BROADER: does not compile
error: load(String) in BadLoader cannot implement
      load(String) in Loader
      overridden method does not throw Exception
}
```

Rule: an overriding method may declare the **same, narrower, or fewer** checked exceptions. Never new or broader ones.

Why: callers code against the **parent type**: `Loader L = new BadLoader(); try { L.load(f); } catch (IOException e) {...}`

That catch was built from the parent's `throws`. A subclass that suddenly throws `Exception` would escape it. Polymorphism must not break the caller's promises.

Unchecked exceptions are not part of the contract, so any override may throw them.

Same rule for a subclass overriding a superclass method.

Class 14 zip → 04_override_throws/ (uncomment BadLoader to see the error)

Six rules for code that fails well

- 1. Validate at the door, fail fast.** Check arguments in constructors and setters and `throw` immediately. A bad value that gets stored is a bug you will find next week.
- 2. Unchecked for bugs, checked for the world.** Programmer mistakes (bad index, null, bad argument) → `RuntimeException` family. Things the caller can plan for (missing file, bad input line, insufficient funds) → checked, possibly your own.
- 3. Catch where you can do something.** Skip the line, tell the user, use a default, retry. If all you can do is print and rethrow, let it propagate instead. Not every method needs a try.
- 4. Never an empty catch.** `catch (Exception e) { }` turns a crash you would see into a wrong answer you will not.
- 5. Exceptions are not an if.** Checking `in.hasNextInt()` beats catching `InputMismatchException`. A thrown exception is slow and means "this should not have happened".
- 6. Say what, where, and why.** Put the file name, line number and the bad value in the message. Keep the original as the **cause**. Your future self is the one reading it.

Gaddis 11.2 covers rules 1, 2 and 6; the rest are conventions you will meet in every Java code base.

Compiles, runs, or crashes? (4 snippets)

A and B

```
// A
static void a() {
    throw new IOException("disk");
}

// B
static void b() throws IOException {
    try {
        throw new IOException("disk");
    } catch (Exception e) {
        System.out.println("caught " + e.getMessage());
    } finally {
        System.out.println("done");
    }
}
// main calls b() inside try/catch(IOException)
```

C and D

```
// C
static int c() {
    try {
        return Integer.parseInt("7");
    } catch (NumberFormatException e) {
        return -1;
    } finally {
        System.out.println("F");
    }
}

// D
class LevelException extends RuntimeException {
    LevelException(int lv) { super("bad " + lv); }
}
static void d() { throw new LevelException(150); }
// main: d(); System.out.println("after");
```

For each: does it compile? If yes, what prints when main calls it? Decide before the next slide.

Compiles, runs, or crashes? Answers

A: does not compile. `IOException` is checked, and `a()` neither catches it nor declares `throws IOException`: *unreported exception java.io.IOException; must be caught or declared to be thrown*.

B: compiles and prints `caught disk then done`. `catch (Exception e)` matches an `IOException` (is-a). The `throws IOException` is allowed even though nothing escapes; it is only a promise.

C: compiles, prints F, returns 7. `parseInt("7")` succeeds, `return 7` waits for `finally`, which prints `F`. The catch never runs.

D: compiles, crashes. `LevelException` is unchecked, so `d()` needs no `throws`. `main` has no catch, so the JVM prints `Exception in thread "main" LevelException: bad 150` and stops. `after` is never printed.

Pattern to notice: **compile errors** come from checked exceptions; **crashes** come from unchecked ones nobody caught.

In-class capstone: Pokédex loader v2 (report, never crash)

```
// PokedexLoader.java: returns the good, keeps the bad
public class PokedexLoader {
    private ArrayList<BadLineException> rejects;
    public ArrayList<BadLineException> getRejects()

    public ArrayList<Pokemon> load(String file)
        throws IOException {          // TODO 3
        BufferedReader br = new BufferedReader(
            new FileReader(file));      // TODO 4
        ...
        dex.add(Pokemon.fromCsv(line)); // TODO 2
        ...
    }
}

// Main.java
dex = loader.load(file);           // TODO 5
for (BadLineException bad : loader.getRejects())
    println(bad.getMessage() + " ("
        + bad.getCause().getClass().getSimpleName() + ")");

// goal:  java Main
// Loaded 7 Pokemon from pokemon_dirty.csv, 4 rejected:
//   line 5 is bad: "25,Pikachu,..." (NumberFormatException...)
//   line 8 is bad: "39,Jigglypuff,..." (ArrayIndex...)
//   ...
// Rejected lines written to rejects.txt
// goal:  java Main pokemon_dirt.csv
// Could not load the Pokedex: cannot open
//   pokemon_dirt.csv (FileNotFoundException)
```

A loader that the GUI or a test calls must not print or exit. It **returns**, **collects**, or **throws**.

TODO 1: run it as-is. One bad line still kills the load.

TODO 2: per line: catch, `rejects.add(new BadLineException(lineNo, line, e))`, go on.

TODO 3: finish `PokedexLoadException(message, cause)`. Catch `IOException` around the file, throw it wrapped. `load` now `throws PokedexLoadException` only.

TODO 4: try-with-resources instead of `br.close()`.

TODO 5: `Main` catches it (message + cause's class), then writes the rejects to `rejects.txt` with a `PrintWriter` (try-with-resources).

Class 14 zip → 05_pokedex_loader_v2_starter/ (BadLineException is given; goal output in Main.java)

Check yourself

1. `main` calls `a()`, `a()` calls `b()`, `b()` throws an unchecked exception. `a()` has a `finally` but no `catch`; `main` has a matching `catch`. List what runs, in order, after the throw.
2. What does `super(message, cause)` do that `super(message)` does not? Which method gets you the original exception back?
3. Rewrite with try-with-resources: `Scanner in = new Scanner(new File(f)); try { ... } finally { in.close(); }`
4. `interface Saver { void save() throws IOException; }` Which of these overrides compile? (a) `void save() throws Exception` (b) `void save()` (c) `void save() throws FileNotFoundException` (d) `void save() throws IllegalStateException`
5. A method catches `NumberFormatException`, prints "oops", and returns 0 as the value. Which design rule does that break, and what should it do instead?
6. In the capstone, why does `load` throw `PokedexLoadException` for a missing file but **not** for a bad line?

Check yourself: answers

1. `b()` ends at once → `a()`'s `finally` runs, then `a()` ends → `main`'s `catch` runs → `main` continues after the try statement.
2. It stores the original exception inside the new one. `getCause()` returns it; `printStackTrace()` shows it under `Caused by:`.
3. `try (Scanner in = new Scanner(new File(f))) { ... }` and delete the `finally`. The `FileNotFoundException` still needs a catch or `throws`.
4. **(b), (c), (d)** compile. (b) fewer, (c) narrower (`FileNotFoundException` is-a `IOException`), (d) unchecked, never restricted. **(a)** is broader than `IOException`: rejected.
5. Rule 4/6: it hides the failure behind a fake value (0 looks like a real level). Either skip the entry and **count** it, or wrap and re-throw with the bad text in the message, so the caller decides.
6. A bad **line** is expected and the rest of the file is still useful: collect it and go on. A missing **file** means nothing can be loaded: there is no partial result to return, so throw.

Wrap-up & what's next

Propagation: an exception travels up the call stack until a matching `catch`; every method it passes through ends, after running its `finally`. **Where** you catch decides how much work is thrown away.

Wrap and re-throw: `throw new MyException("what it means here", e);` `getCause()` and `Caused by:` keep the original.

try-with-resources: `try (Reader r = ...) { }` closes for you, in reverse order, exception or not. Use it for every file from now on.

Overriding: same, narrower, or fewer checked exceptions; never broader. Unchecked is free.

Practice: finish the capstone (Class 14 zip → `05_pokedex_loader_v2_starter`, TODO 1–5). **HW3 Q1–Q4** use everything from today.

Next, Thu Oct 29 (Class 16): Chapter 11.3: binary files, random-access files, and object serialization (HW3 Q5). Read Gaddis 11.3.

Sources

Gaddis, **Starting Out with Java**, 4th ed., Ch. 11.1 (the `finally` clause, the stack trace, exception propagation) and Ch. 11.2 (throwing exceptions, creating exception classes); Ch. 11.3 is Thursday.

Java SE 17 API: `java.lang.Throwable` (`Throwable(String, Throwable)`, `getCause`), `AutoCloseable`; `java.io.BufferedReader`, `PrintWriter`, `FileNotFoundException`, `IOException`. Java Language Specification §14.20.3 (try-with-resources) and §8.4.8.3 (overriding and `throws`).

`Pokemon`, `pokemon_dirty.csv` and `BadLineException` are the same as Classes 12–14's examples; the `setLevel` solution is Class 13's in-class exercise.

All outputs and compiler errors shown were checked with OpenJDK 17.