

Class

14

CSC 240 Computer Science III

Exceptions II–III: throw vs throws, Your Own Exceptions & Propagation

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 9 · Thursday, October 22, 2026
wcu.ninja/csc240

Where we left off, and today's plan

Tuesday we got through: the Pokédex loader, `finally` (always runs), and `throw` in the `Circle` constructor.

Today, part 1 (Class 13's second half): `throw` vs `throws`, which built-in exception to throw, writing your **own** exception class, checked or unchecked → in-class: a Pokémon that refuses a bad level.

Today, part 2 (Class 14): how an exception travels up the call stack, wrapping it with a **cause**, try-with-resources, and six design rules.

HW3 (Warehouse) is out today, due **Tue Nov 10, 11:59 PM**. Everything in it is covered by the end of today, except Persistence (Class 15–16).

Code: Class 13 zip (folders 02–04) and Class 14 zip (folders 01–03), both on `wcu.ninja`.

throw: you can throw an exception too

```
public class Circle {           // Class 7's Circle
    private double radius;

    public Circle(double radius) {
        if (radius <= 0) {
            throw new IllegalArgumentException(
                "radius must be positive, got " + radius);
        }
        this.radius = radius;    // never reached on throw
    }
    ...
}

// main:
double[] radii = {2.0, -1.0, 0.0, 3.5};
for (double r : radii) {
    try {
        Circle c = new Circle(r); // may throw
        System.out.println("made " + c);
    } catch (IllegalArgumentException e) {
        System.out.println("rejected: " + e.getMessage());
    }
}

// made Circle(r=2.0, area=12.57)
// rejected: radius must be positive, got -1.0
// rejected: radius must be positive, got 0.0
// made Circle(r=3.5, area=38.48)
```

`throw new`

`SomeException("message");` is a statement. It creates the exception object and throws it, **right there**.

The method (or constructor) ends at the `throw`. For a constructor that means **no object is created**: a Circle with radius -1 never exists.

`IllegalArgumentException` is the standard choice for "you passed me a bad value". Unchecked: the caller is not forced to catch it, but can.

The old habit, `if (radius > 0)` `this.radius = radius;`, silently keeps the old value and hides the bug. **Fail fast, fail loudly.**

Class 13 zip → 02_throw/ (Circle.java, Main.java)

throw vs. throws (one letter, two different things)

throw (a statement)

Lives **inside** a method body.

```
throw new  
IllegalArgumentException("...");
```

Throws **one** exception object, right now.

Ends the method immediately. Nothing after it runs (except `finally`).

Works for checked and unchecked exceptions alike.

Think: "I am raising the alarm."

throws (a clause)

Lives in the method **header**, after the parameter list.

```
int countLines(String f) throws  
FileNotFoundException {
```

Lists **types** the method might let escape. Throws nothing itself.

Required only for **checked** exceptions (`IOException` family, and your own `extends Exception`).

Pushes the decision to the caller: catch it, or declare it again.

Think: "Warning: this might go off."

`throw new IOException(...)` in the body needs `throws IOException` in the header. So does a checked call.

Which exception should you throw? (built-in choices)

```
public void withdraw(double amount) {
    if (amount <= 0) {
        throw new IllegalArgumentException(
            "amount must be positive, got " + amount);
    }
    if (amount > balance) {
        throw new IllegalStateException(
            "insufficient funds: balance " + balance
            + ", requested " + amount);
    }
    balance -= amount;
}
```

```
// main, balance 100, requests 30, -5, 500, 70:
// withdrew 30.0, balance 70.0
// bad request: amount must be positive, got -5.0
// declined: insufficient funds: balance 70.0,
//           requested 500.0
// withdrew 70.0, balance 0.0
```

```
// then acct.withdraw(1000) with NO catch:
Exception in thread "main"
    java.lang.IllegalStateException: insufficient
    funds: balance 0.0, requested 1000.0
    at BankAccount.withdraw(BankAccount.java:16)
    at Main.main(Main.java:32)
```

`IllegalArgumentException`: the **argument** is wrong (negative amount, radius 0).

`IllegalStateException`: the argument is fine, but the **object** is not in a state where this call makes sense (withdraw 500 from 70).

`NumberFormatException` is-a `IllegalArgumentException`: `parseInt` does what you do here.

Both are unchecked: no `throws` needed, the caller may catch. Uncaught, it prints the trace and stops.

Put the **value** in the message. "bad amount" is useless at 2 AM; "got -5.0" is not.

No built-in name fits? Write your own (next).

Class 13 zip → 02_throw/ (BankAccount.java)

Your own exception class: it is just a class

```
// InsufficientFundsException.java
public class InsufficientFundsException
    extends Exception {    // CHECKED
    private double shortBy;

    public InsufficientFundsException(double shortBy) {
        super("insufficient funds: short by $" + shortBy);
        this.shortBy = shortBy;
    }

    public double getShortBy() { return shortBy; }
}

// That is the whole file. Nothing else is required.

// Chapter 10 again:
//   Throwable
//   +-- Exception
//       +-- InsufficientFundsException    <- yours
//       +-- RuntimeException
//       +-- IllegalArgumentException
```

`extends Exception` is what makes it an exception. `getMessage()`, `printStackTrace()`, propagation: all **inherited** from `Throwable`.

`super(message)` stores the text that `getMessage()` returns. You choose the wording once, here.

Add fields for anything the catcher might want to **use**, not just print: the shortfall, a line number, a file name.

Naming convention: ends in `Exception`.

Why bother? `catch`

`(InsufficientFundsException e)` says exactly what happened.

`catch (RuntimeException e)` could be anything.

Class 13 zip → 03_custom_exception/InsufficientFundsException.java

Using it: throws in the method, catch in the caller

```
// BankAccount.java
public void withdraw(double amount)
    throws InsufficientFundsException { // REQUIRED
    if (amount <= 0)
        throw new IllegalArgumentException(
            "amount must be positive, got " + amount);
    if (amount > balance)
        throw new InsufficientFundsException(amount - balance);
    balance -= amount;
}

// Main.java
for (double amt : requests) {
    try {
        acct.withdraw(amt);
        System.out.println("withdrew " + amt);
    } catch (InsufficientFundsException e) {
        System.out.println("declined: " + e.getMessage()
            + " (deposit at least $" + e.getShortBy()
            + " first)");
    } catch (IllegalArgumentException e) {
        System.out.println("bad request: " + e.getMessage());
    }
}

// withdrew 30.0
// declined: insufficient funds: short by $430.0
//      (deposit at least $430.0 first)
// bad request: amount must be positive, got -5.0
```

Checked, so the compiler enforces the contract: remove `throws` from `withdraw` and javac says **unreported exception `InsufficientFundsException`; must be caught or declared to be thrown.**

Remove the `catch` in `main`: same error there. **Somebody must handle it.**

`IllegalArgumentException` needs no `throws`: unchecked.

The catch uses `e.getShortBy()`: data travelled from the throw to the catch inside the object. That is the payoff.

Two unrelated types, so the catch order does not matter here.

Class 13 zip → 03_custom_exception/

Checked or unchecked for your own class?

Exception → **checked**

The caller **must** catch or declare. The compiler will not let them forget.

Use when the caller can reasonably **recover** or explain: insufficient funds, missing file, a user typed a bad level.

Cost: `throws` clauses everywhere on the path from the throw to the catch.

Gaddis's custom exceptions are all checked. **Default choice in this course.**

RuntimeException → **unchecked**

The caller **may** catch. No `throws` needed anywhere.

Use for **programmer mistakes** the caller should have prevented: negative index, null where an object is required, a bad argument.

Cost: nothing forces anyone to handle it, so it can surface as a crash far from the cause.

The built-ins you throw by hand (`IllegalArgumentException`, `IllegalStateException`) are this kind.

Rule of thumb: "could a careful caller have avoided this?" Yes → unchecked. No (the world did it) → checked.

In-class: a Pokémon that refuses a bad level

```
// Pokemon.java (Classes 5-12), near the bottom:
public void setLevel(int level) {
    if (level >= 1 && level <= 100) this.level = level;
} // a bad level is SILENTLY IGNORED

// Main.java: the trainer typed these by hand
String[] entries = {"Pikachu:25", "Mewtwo:150",
    "Eevee:twelve", "Snorlax:100", "Charizard:-3"};
int rejected = 0;
for (String entry : entries) {
    String[] parts = entry.split(":");
    Pokemon p = find(dex, parts[0]);
    int level = Integer.parseInt(parts[1]); // TODO 4
    p.setLevel(level); // TODO 2-4
    System.out.println(p.getName() + " -> Lv." + p.getLevel());
}

// goal:
// 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
```

TODO 1: run it. Mewtwo quietly stays Lv.1 (why?), then `Eevee:twelve` crashes the loop.

TODO 2: in `setLevel`, replace the silent `if` with `throw new IllegalArgumentException(...)`. What happens to Snorlax now?

TODO 3: new file `InvalidLevelException` extends `Exception`: takes the bad level, calls `super(...)`, has `getLevel()`. Make `setLevel` throw **that**. Read javac's complaint. Fix it.

TODO 4: catch it in the loop, print, count. Keep a catch for `NumberFormatException` too.

TODO 5 (think): checked or unchecked here? What changes in `Main` if you switch?

Class 13 zip → 04_pokemon_level_starter/ (javac *.java && java Main)

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 Tuesday: `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/

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.

Wrap-up & what's next

throw (statement, in the body) raises one exception now; **throws** (clause, in the header) is required only for checked exceptions.

Your own exception: `extends Exception` (checked, caller must handle) or `RuntimeException` (unchecked, for bugs); call `super(message)`; add fields the catcher can use.

Propagation: an exception leaves every method between the `throw` and the matching `catch`, running each `finally` on the way. Where you catch decides what is lost.

Wrap and re-throw with `super(message, cause)`; read `Caused by:` from the bottom.
try-with-resources for every file from now on.

Practice: finish `04_pokemon_level_starter` (Class 13 zip). Optional: the Pokédex loader v2 capstone (Class 14 zip, `05_...`), it is a small HW3.

HW3 due Tue Nov 10. **Next, Tue Oct 27 (Class 15):** we start with overriding + `throws` (5 min), then Ch. 11.3: binary files, random access, serialization. Read Gaddis 11.3.

APPENDIX

Not covered in class today

Overriding and throws opens Class 15. The Pokédex loader v2 is optional practice that uses everything from today.

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)

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)

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 next week.

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.

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