

Class

13

CSC 240 Computer Science III

Exceptions II: finally, throw & Your Own Exception Classes

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 9 · Tuesday, October 20, 2026
wcu.ninja/csc240

Start here: the Pokédex loader that survives

```
// pokemon_dirty.csv was typed by hand...
25,Pikachu,Electric,,35,fifty-five,40,90,false
39,Jigglypuff,Normal,Fairy,115,45,20
#133,Eevee,Normal,,55,55,50,55,false
150,Mewtwo,Psychic,,106,110,90,,true

// Main.java starter: 1 bad line = no Pokedex
while ((line = br.readLine()) != null) {
    lineNo++;
    if (line.trim().isEmpty()) continue;
    dex.add(Pokemon.fromCsv(line)); // TODO
}

// goal (full text in Main.java):
// Line 5 skipped: ... "fifty-five"
// Line 8 skipped: Index 7 out of bounds ...
// Line 10 skipped: ... "#133"
// Line 13 skipped: ... ""
// Loaded 7 Pokemon, skipped 4 bad lines.
```

We ran out of time for this on Thursday. Do it now.

TODO 1: run it as-is. Which exception, which CSV line?

TODO 2: try/catch around `fromCsv`. Print the message and count skips.

TODO 3: a second exception type appears. Add a catch or use `|`.

TODO 4: misspell the file name. Print "Cannot open ..." instead of a trace.

TODO 5: why is `#133` already handled?

Class 12 zip → 05_pokedex_loader_starter/ (run from inside the folder)

Pokédex loader: one way to write it

```
public static void main(String[] args) {
    ArrayList<Pokemon> dex = new ArrayList<>();
    int skipped = 0;
    try {
        // TODO 4: around the FILE
        BufferedReader br =
            new BufferedReader(new FileReader(FILE));
        br.readLine();
        int lineNo = 1; String line;
        while ((line = br.readLine()) != null) {
            lineNo++;
            if (line.trim().isEmpty()) continue;
            try {
                // TODO 2: around ONE LINE
                dex.add(Pokemon.fromCsv(line));
            } catch (NumberFormatException
                | ArrayIndexOutOfBoundsException e) {
                System.out.println("Line " + lineNo
                    + " skipped: " + e.getMessage());
                skipped++;
                // TODO 3: multi-catch
            }
        }
        br.close();
    } catch (FileNotFoundException e) {
        System.out.println("Cannot open " + FILE
            + " - check the file name.");
        return;
    } catch (IOException e) { // the general one LAST
        System.out.println("Error reading " + FILE);
        return;
    }
    // ... print Loaded/skipped, sort, Strongest
}
```

Two try blocks, two jobs. The inner one protects one CSV line, so a bad line skips only itself. The outer one protects the whole file.

`FileNotFoundException` is-a `IOException`, so the specific catch comes first (Class 12).

`return` inside the catch: no file, nothing else to do.

TODO 5: `"#133"` goes through `Integer.parseInt`, which throws `NumberFormatException`, already caught by TODO 2.

Thursday we rebuild this as a `PokedexLoader` class that **reports** instead of printing.

Same output as the goal on the previous slide. Keep your own version; there is more than one right answer.

TODAY

Throwing, Not Just Catching

Class 12 was about catching exceptions that other code throws. Today: a block that always runs, throwing an exception yourself when a caller hands you garbage, and defining exception classes of your own.

finally: the block that always runs

```
static int parseLevel(String entry) {
    System.out.println(" try: parsing " + entry);
    try {
        int level = Integer.parseInt(
            entry.split(":")[1].trim());
        System.out.println(" try: got " + level);
        return level;    // finally STILL runs first
    } catch (NumberFormatException e) {
        System.out.println(" catch: " + e.getMessage());
        return -1;
    } finally {
        System.out.println(" finally: done with " + entry);
    }
}

// main:
parseLevel("Pikachu:25");    // case 1: no exception
parseLevel("Mewtwo:seventy"); // case 2: caught
try {
    parseLevel("Eevee");    // case 3: NOT caught
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("main caught: " + e.getMessage());
}
```

`finally { }` comes after the last `catch`. It runs **no matter how the try block ends**: normally, by `return`, by a caught exception, or by one that propagates.

Even `return level;` does not skip it. Java remembers the return value, runs `finally`, then returns.

A `try` needs at least one `catch` or a `finally`. `try { } finally { }` with no `catch` is legal.

What is it for? Cleanup that must happen either way: closing a file, a network connection, a database.

Not for "the program ends normally" logic. That goes after the whole `try` statement.

Class 13 zip → 01_finally/Main.java

Trace it: three cases, one finally

Case 1: no exception

```
try: parsing Pikachu:25
try: got 25
finally: done with Pikachu:25
returned 25
```

Case 2: exception, caught

```
try: parsing Mewtwo:seventy
catch: For input string: "seventy"
finally: done with Mewtwo:seventy
returned -1
```

Case 3: exception, NOT caught here

```
try: parsing Eevee
finally: done with Eevee
main caught: Index 1 out of bounds for length 1
Program ends normally.
```

Case 1: try runs to `return level`. `finally` prints, then 25 comes back.

Case 2: `parseInt` throws, the rest of the try is skipped, the catch prints, `finally` prints, then -1 comes back.

Case 3: `"Eevee".split(":")[1]` throws an `ArrayIndexOutOfBoundsException`. No catch in `parseLevel` matches it, so the method ends... but `finally` **still runs on the way out**, and the exception keeps travelling up to `main`.

Order to remember: **try** → **(catch)** → **finally** → **continue or propagate**.

Run it: `cd 01_finally && javac Main.java && java Main`

The classic job for finally: close the file

```
static int countNumeric(String fileName)
    throws FileNotFoundException {
    Scanner in = new Scanner(new File(fileName));
    int count = 0;
    try {
        while (in.hasNextLine()) {
            Integer.parseInt(in.nextLine().trim());
            count++;           // may throw NumberFormat...
        }
        return count;
    } finally {
        in.close();           // on return AND on throw
        System.out.println(" (file closed)");
    }
}

// main, for levels_ok.txt / levels_bad.txt / missing.txt:
// (file closed)
// levels_ok.txt: 3 numbers
// (file closed)
// levels_bad.txt: bad line For input string: "seventy"
// missing.txt: not found
```

An open file is a resource the operating system lends you. Leave it open and later writes may be lost, or the file stays locked.

Without `finally`, the bad line in `levels_bad.txt` would throw **past** `in.close()` and the file would never be closed.

The `Scanner` is created **before** the try: if the file is missing there is nothing to close, and the `FileNotFoundException` goes straight to the caller.

No `catch` here at all: this method does cleanup, `main` does the explaining.

Thursday: try-with-resources does the `close()` for you.

Class 13 zip → 01_finally/CountLines.java (run from inside the folder)

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)

Check yourself

1. What prints? `try { print("A"); return; } finally { print("B"); } print("C");` (inside a method called from main)
2. A constructor does `throw new IllegalArgumentException("bad")` on its first line. Does the object get created? Is `throws` needed in the constructor header?
3. Fill in the blank: `public void withdraw(double amt) _____ InsufficientFundsException { ... _____ new InsufficientFundsException(amt - balance); }`
4. `class BadLevelException extends RuntimeException { ... }` Does a method that throws it need a `throws` clause? Must the caller catch it?
5. Why does `super("level must be 1-100, got " + level);` matter? What would `e.getMessage()` return without it?
6. Name a situation where `finally` runs but no `catch` does.

Check yourself: answers

1. **A B** and then C only if the method's caller continues. Inside the method, `return` triggers `finally` (B) and then leaves, so `print("C")` in that method never runs.
2. **No object** is created: the constructor ended at the `throw`. **No `throws`** is needed: `IllegalArgumentException` is unchecked.
3. `throws` in the header, `throw` in the body. One letter apart, two different jobs.
4. It extends `RuntimeException`, so it is **unchecked**: no `throws` clause required, and the caller is **not** forced to catch it (but may).
5. `super(message)` stores the text that `getMessage()` returns. Without it, `getMessage()` returns `null` and your catch prints "rejected: null".
6. The try block completes normally (no exception), or it throws something **no catch in that method matches**: `finally` still runs, then the exception keeps propagating.

Wrap-up & what's next

finally runs no matter how the try ends: normal, `return`, caught, or propagating. Use it for cleanup (close the file).

throw new X("message with the bad value") ends the method right there. In a constructor, no object is created. Prefer `IllegalArgumentException` / `IllegalStateException` for bad input and bad state.

throw is a statement in the body; **throws** is a clause in the header, required only for checked exceptions.

Your own exception = a class that `extends` `Exception` (checked) or `RuntimeException` (unchecked), calls `super(message)`, and may carry extra fields.

Practice: finish `04_pokemon_level_starter` TODO 1–5 if you did not in class. Expected output is at the top of `Main.java`.

Next, Thu Oct 22 (Class 14): where to catch and what gets skipped (propagation + finally together), wrapping an exception with a **cause**, **try-with-resources**, overriding methods that throw, and designing a loader that reports instead of crashes. Read Gaddis 11.2.

Sources

Gaddis, **Starting Out with Java**, 4th ed., Ch. 11.1 (the `finally` clause, checked and unchecked exceptions) and Ch. 11.2 (throwing exceptions, creating your own exception classes).

Java SE 17 API: `java.lang.Throwable` (constructors, `getMessage`), `Exception`, `RuntimeException`, `IllegalArgumentException`, `IllegalStateException`; `java.util.Scanner`; `java.io.FileNotFoundException`.

`Circle` is Class 7's; `Pokemon` and the Pokédex data are the same as Classes 5–12.

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