

Class

12

CSC 240 Computer Science III

Exceptions: try / catch, the Hierarchy & Checked vs. Unchecked

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 8 · Thursday, October 15, 2026
wcu.ninja/csc240

Warm-up: you have already met exceptions

- **Class 4:** `Integer.parseInt("ninety")` stopped with a `NumberFormatException`.
- **Class 10:** the fleet loader caught a bad axle count with `catch (NumberFormatException e)`.
- **HW2:** `public static void main(String[] args) throws IOException`. What does that `throws` actually mean?
- **Q:** your Pokédex CSV has 800 good lines and **one typo**. Today the program loads... how many Pokémon?
- Today: what an exception is, how to read one, and how to **handle** it instead of crashing. Chapter 11 starts now.

Welcome back from Fall Break. Chapter 11 starts the second half of the course.



TODAY

When Things Go Wrong

An exception is an object that says "something went wrong here". Java throws it, and it travels back up the chain of method calls until some code catches it. If nobody catches it, the program stops and prints a stack trace.

An exception crashes the program

```
static int parseLevel(String entry) {
    String[] parts = entry.split(":");
    return Integer.parseInt(parts[1].trim());
}

static int totalLevels(String[] team) {
    int total = 0;
    for (String entry : team) {
        int level = parseLevel(entry);
        System.out.println("  " + entry
            + " -> level " + level);
        total += level;
    }
    return total;
}

// main:
String[] team = {"Pikachu:25", "Charizard:36",
    "Mewtwo:seventy", "Eevee:12"};
int total = totalLevels(team);
System.out.println("Total = " + total);
// ^ never runs
```

- `parseInt("seventy")` cannot return a number, so it **throws** a `NumberFormatException` object.
- Nobody catches it. `parseLevel` ends, then `totalLevels` ends, then `main` ends.
- Eevee is never read. `Total` is never printed.
- One bad entry and **all** the work is lost.

Class 12 zip → 01_stack_trace/

Reading a stack trace

```
Adding up the team's levels...
Pikachu:25 -> level 25
Charizard:36 -> level 36
Exception in thread "main"
    java.lang.NumberFormatException:
        For input string: "seventy"
        at java.base/...forInputString(...)
        at java.base/...Integer.parseInt(...)
        at java.base/...Integer.parseInt(...)
        at Main.parseLevel(Main.java:12)
        at Main.totalLevels(Main.java:18)
        at Main.main(Main.java:28)
```

Read the trace from the top. Don't guess.

- **Line 1: what** went wrong: the exception class and its message.
- **"at" lines: where.** Newest call on top, `main` at the bottom.
- Skip the `java.base/...` lines. That is library code, not your bug.
- Start at the **first line from your own file:** `Main.java:12`.
- Try it: change `"Mewtwo:seventy"` to `"Mewtwo"`. Which exception do you get now?

try / catch: handle it and keep going

```
for (String entry : team) {
    try {
        int level = parseLevel(entry);
        System.out.println("  " + entry
            + " -> level " + level);
        total += level;
    } catch (NumberFormatException e) {
        System.out.println("  " + entry
            + " -> skipped (" + e.getMessage() + ")");
        skipped++;
    }
}
System.out.println("Total = " + total
    + ", skipped " + skipped);

// Pikachu:25 -> level 25
// Charizard:36 -> level 36
// Mewtwo:seventy -> skipped
//   (For input string: "seventy")
// Eevee:12 -> level 12
// Total = 73, skipped 1
```

- `try { }` holds code that **might** throw.
- `catch (Type e) { }` runs **only if** an exception of that type (or a subclass) was thrown.
- If a line in `try` throws, the **rest of the try block is skipped** and Java jumps to the catch.
- After the catch, the loop goes on to Eevee.
- No exception means the catch block never runs.

Class 12 zip → 02_try_catch/

Trace the flow: which lines run?

entry = "Charizard:36" (no exception)

```
try {  
    level = parseLevel(entry); // 1 ✓  
    print(...);                // 2 ✓  
    total += level;             // 3 ✓  
} catch (NumberFormatException e) {  
    print("skipped");           //   ✗  
    skipped++;                  //   ✗  
}  
nextStatement();               // 4 ✓
```

entry = "Mewtwo:seventy" (throws)

```
try {  
    level = parseLevel(entry); // 1 ✗ throws  
    print(...);                //   skipped  
    total += level;             //   skipped  
} catch (NumberFormatException e) {  
    print("skipped");           // 2 ✓  
    skipped++;                  // 3 ✓  
}  
nextStatement();               // 4 ✓
```

Either way the program reaches nextStatement(). Without try/catch, the right side would end the program at step 1.

The exception is an object

```
try {
    Integer.parseInt("12x");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    System.out.println(
        e.getClass().getSimpleName());
    // e.printStackTrace();
}
System.out.println("Still running.");

// For input string: "12x"
// java.lang.NumberFormatException:
//     For input string: "12x"
// NumberFormatException
// Still running.
```

- `e` is a reference to the exception object that was thrown, like a parameter.
- `getMessage()`: the explanation, good for messages to the user.
- `toString()`: the class name plus the message.
- `printStackTrace()`: the full trace, but the program **keeps going**. Useful while debugging.
- Exceptions are classes, so they have a **hierarchy** (next slide).

Class 12 zip → 02_try_catch/

Exceptions are classes in a hierarchy

```
Throwable
+-- Error          // don't catch
|   +-- OutOfMemoryError
+-- Exception      // checked
    +-- IOException
        |   +-- FileNotFoundException
    +-- RuntimeException // unchecked
        +-- IllegalArgumentException
            |   +-- NumberFormatException
        +-- IndexOutOfBoundsException
            |   +-- ArrayIndexOutOfBoundsException
        +-- NullPointerException
        +-- ArithmeticException
```

- Every exception class **extends** `Throwable`. This is Chapter 10 inheritance again.
- A `NumberFormatException` **is-a** `IllegalArgumentException`, which **is-a** `RuntimeException`, which **is-a** `Exception`.
- So `catch (RuntimeException e)` also catches a `NumberFormatException`. That is **polymorphism**.
- You catch things under `Exception`. You don't catch `Error`.

Gaddis 11.1, "Exception classes"

Several catch blocks: most specific first

```
String[] team = {"Pikachu:25",  
    "Mewtwo:seventy", "Eevee", null};  
for (String entry : team) {  
    try {  
        System.out.println(entry + " -> level "  
            + parseLevel(entry));  
    } catch (NumberFormatException e) {  
        System.out.println(entry + " -> not a number");  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println(entry + " -> no ':'");  
    } catch (NullPointerException e) {  
        System.out.println("null -> missing");  
    }  
}  
// Pikachu:25 -> level 25  
// Mewtwo:seventy -> not a number  
// Eevee -> no ':'  
// null -> missing
```

- One `try` can have many `catch` blocks.
- Java checks them **top to bottom** and runs the **first one that matches**. At most one catch runs.
- `"Eevee".split(":")` has length 1, so `parts[1]` throws.
- `null.split(...)` throws a `NullPointerException`.
- A superclass catch above a subclass catch **does not compile** (next slide).

Class 12 zip → 03_catch_order/

Multi-catch, and why order matters

```
// two types, one handler: join with |
try {
    System.out.println(parseLevel(entry));
} catch (NumberFormatException
        | ArrayIndexOutOfBoundsException e) {
    System.out.println("bad entry: "
        + e.getClass().getSimpleName());
} catch (RuntimeException e) {
    System.out.println("other problem");
}

// superclass FIRST: does not compile
try {
    parseLevel("Eevee");
} catch (RuntimeException e) {
    System.out.println("general");
} catch (NumberFormatException e) {
    System.out.println("specific");
}
error: exception NumberFormatException
       has already been caught
```

- `catch (A | B e)`: one block handles both types.
- A superclass catch (`RuntimeException`) at the **end** is a safety net for anything else.
- If the superclass comes first, the subclass catch could never run, so javac rejects it.
- Avoid `catch (Exception e) { }` with an empty body. It hides bugs.

Class 12 zip → 03_catch_order/

Checked vs. unchecked exceptions

Unchecked (RuntimeException)

- Usually a **bug in your code**: bad index, `null`, bad number format
- `NumberFormatException`, `ArrayIndexOutOfBoundsException`, `NullPointerException`, `ArithmeticException`
- The compiler does **not** make you handle them
- Best fix: usually a check (`if`) or a `try/catch` around input you don't control

Checked (every other Exception)

- Something **outside** your program failed: missing file, disk error, network
- `IOException`, `FileNotFoundException`
- The compiler **forces** you to choose: **catch it** or **declare it** with `throws`
- Otherwise: "unreported exception ...; must be caught or declared to be thrown"

Checked: catch it, or declare it with throws

```
// (b) DECLARE: the caller has to deal with it
static int countLines(String fileName)
    throws FileNotFoundException {
    Scanner in = new Scanner(new File(fileName));
    int count = 0;
    while (in.hasNextLine()) {
        in.nextLine(); count++;
    }
    in.close();
    return count;
}

public static void main(String[] args) {
    String[] files = {"team.txt", "tem.txt"};
    for (String name : files) {
        try {
            // (a) CATCH
            System.out.println(name + ": "
                + countLines(name) + " lines");
        } catch (FileNotFoundException e) {
            System.out.println(name + ": not found");
        }
    }
}

// team.txt: 4 lines
// tem.txt: not found
```

- `new Scanner(new File(...))` may throw `FileNotFoundException`, a **checked** exception.
- `throws` in the header means: "I might throw this. **Caller**, be ready."
- Remove `throws` and javac reports **unreported exception**.
- Catch it where you know **what to tell the user**. Here that is `main`.
- HW2's `main(...) throws IOException` passes it to the JVM, which prints the trace.

Class 12 zip → 04_checked_throws/ (run from inside the folder)

The exception travels up the call stack

```
main()                // catch? yes: handled
+-- totalLevels()      // catch? no: pass up
    +-- parseLevel()   // catch? no: pass up
        +-- parseInt() // THROWS

// Java looks for a matching catch:
// 1. in the method that threw
// 2. then in its caller, and so on up to main
// 3. none at all: JVM prints the trace, stops

// ONE try/catch in the right place protects
// every call below it:
try {
    total = totalLevels(team);
} catch (NumberFormatException e) {
    System.out.println("Bad level in the team");
}
```

- This upward search is called **propagation**. It also explains the order of the stack trace.
- Each method the exception passes through **ends immediately**.
- **Where** you catch decides **what** gets skipped: one entry (catch inside the loop) or the whole team (catch around the call).
- **throws** on a method just documents this for checked exceptions.

Gaddis 11.1, "Handling exceptions" and "The stack trace"

In-class: a 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.
```

- **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)



Check yourself

- 1. Checked or unchecked? `NumberFormatException`, `FileNotFoundException`, `NullPointerException`, `IOException`
- 2. What prints? `try { print("A"); Integer.parseInt("x"); print("B"); } catch (NumberFormatException e) { print("C"); } print("D");`
- 3. Why doesn't this compile? `catch (Exception e) {...} catch (IOException e) {...}`
- 4. A method calls `new Scanner(new File(name))` with no try/catch, and javac complains. Name **two** ways to fix it.
- 5. In the `01_stack_trace` output, which line should you open first, and why?

Check yourself: answers

- 1. Checked: `FileNotFoundException`, `IOException`. Unchecked (subclasses of `RuntimeException`): `NumberFormatException`, `NullPointerException`.
- 2. **A C D**. The throw skips the rest of the try (B), the catch prints C, then the program goes on to D.
- 3. `IOException` is a subclass of `Exception`, so the first catch already catches it. The second could never run: "has already been caught".
- 4. (a) Wrap it in `try { ... } catch (FileNotFoundException e) { ... }`. (b) Add `throws FileNotFoundException` to the method header and let the caller catch it.
- 5. `at Main.parseLevel(Main.java:12)`. It is the first line from your own code. The lines above it are inside the Java library.

Wrap-up & what's next

- **Exception** = an object that describes an error. It is **thrown**, travels **up the call stack**, and is either **caught** or ends the program with a stack trace.
- **try/catch**: the rest of the try is skipped, the **first matching** catch runs, and the program goes on. Order the catches from specific to general.
- **Unchecked** (`RuntimeException`): usually bugs. **Checked** (`IOException...`): you must **catch or declare** (`throws`).
- **Practice**: if you did not finish TODO 1–5 of `05_pokedex_loader_starter` in class, finish it before next class.
- **Next: Tue Oct 20, Class 13**: `finally`, `throw`, and writing your own exception classes. Read Gaddis 11.1–11.2.

Sources

- Gaddis, **Starting Out with Java**, 4th ed., Ch. 11.1: handling exceptions, exception classes, multiple catch clauses, the stack trace, checked and unchecked exceptions.
- Java SE 17 API: `java.lang.Throwable`, `Exception`, `RuntimeException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`; `java.io.FileNotFoundException`, `IOException`; `java.util.Scanner`.
- Pokemon class and Pokédex data: the same as Classes 5–11. `pokemon_dirty.csv` was made from `pokemon_data.csv` with typos added on purpose.
- All outputs and compiler errors shown were checked with OpenJDK 17.